

ԱՎԵՏԻՍՅԱՆ Ս. Ս. , ԴԱՆԻԵԼՅԱՆ Ս. Վ.

ԻՆՖՈՐՄԱՏԻԿԱ

11-րդ դասարան

ՀԱՆՐԱԿՐԹԱԿԱՆ ԱՎԱԳ ԴՊՐՈՑԻ
ԲՆԱԳԻՏԱՄԱԹԵՄԱՏԻԿԱԿԱՆ ՀՈՍՔԻ ՀԱՄԱՐ

ԵՐԱՇԽԱՎՈՐՎԱԾ Է ՀՀ ԿՐԹՈՒԹՅԱՆ ԵՎ
ԳԻՏՈՒԹՅԱՆ ՆԱԽԱՐԱՐՈՒԹՅԱՆ ԿՈՂՄԵՑ

ՎԵՐԱՀՐԱՏԱՐԱԿՈՒԹՅՈՒՆ

Ե Ր Ե Վ Ա Ն



2 0 1 7

ՀՏԳ- 373.167.1 : 004 (075.3)
ԳՄԴ- 73 ց 72
Ա 791

Մասնագիտական խմբագիր՝ Ռ. Աղզաշյան

Ավետիսյան Ս.Ս.

Ա 791 Ինֆորմատիկա: 11-րդ դաս. դասագիրք. հանրակրթական ավագ դպրոցի
բնագիտամաթեմատիկական հոսքի համար / Ս. Ս. Ավետիսյան,
Ս. Վ. Դանիելյան, Մասն. խմբ.՝ Ռ. Աղզաշյան. -
Եր.: Տիգրան Մեծ, 2017. - 192 էջ:

ՀՏԳ- 373.167.1 : 004 (075.3)
ԳՄԴ- 73 ց 72

ISBN 978-99941-0-425-3

© Ավետիսյան Ս., Դանիելյան Ս., 2017 թ.
© «Տիգրան Մեծ», 2017 թ.
© ԳՏՀՏՀ, 2017 թ.

ՆԵՐԱԾՈՒԹՅՈՒՆ

Արդեն գիտեք, որ համակարգչում ինֆորմացիան ներկայացվում է գրոների և մեկերի միջոցով: Հաշվիչ տեխնիկայի զարգացման նախնական փուլում (անցած դարի 40-50-ական թվականներին) համակարգչին տրվող հրամանների հաջորդականությունն ի սկզբանե ձևակերպվում էր մեքենային հասկանալի գրոների ու մեկերի, այլ խոսքով՝ **մեքենայական կոդերի** միջոցով:

Մեքենայական կոդով տրված հրամանը բաղկացած է հրամանի (գործողության) կոդից և հրամանի մաս կազմող **օպերանդների** հասցեներից:

Բնականաբար, մեքենայական կոդով աշխատելը աշխատատար և ժամանակ պահանջող գործընթաց էր. անհրաժեշտ էր այնպիսի միջոց գտնել, որը մարդ-մեքենա երկխոսությունը կդարձներ մատչելի: Այդ նպատակով **ծրագրավորման փայրեր լեզուներ** մշակվեցին:

Ծրագրավորման լեզուները համակարգչի հետ հաղորդակցվելու նպատակով սրեղծված արհեստական լեզուներ են, որոնք քերականության ու շարահյուսության ուրույն և խիստ կանոններ ունեն:

20-րդ դարի 50-ական թվականների առաջին կեսերին **Ասեմբլեր** (*Assembly language*) անվամբ լեզուներ ստեղծվեցին, որոնք, գրոներից և մեկերից բացի, նաև մարդկային լեզվին համահունչ (*ADD, SUB, ...* և այլն) հրամաններ, այլ խոսքով, **օպերատորներ** էին ներառում. այստեղ առաջին անգամ մտցվեց **փոփոխական** հասկացությունը՝ որպես տարբեր արժեքներ կրելու հատկությամբ օժտված մեծություն:

Ասեմբլերով գրված ծրագիրը համակարգչին հասկանալի դարձնելու նպատակով առաջին անգամ ստեղծվեց նաև հատուկ թարգմանիչ ծրագիր՝ **Ասեմբլեր լեզվի կոմպիլյատորը**:

Կոմպիլյատորները ծրագրավորման կոնկրետ լեզվով գրված ծրագրի թարգմանման նպատակով սրեղծված ծրագրային միջոցներ են, որոնք նախ սրուզում են գրված ծրագրի քերականությունն ու շարահյուսությունը, ապա, սխալ չհայտնաբերելու դեպքում, այն վերածում (թարգմանում) մեքենայական կոդի:

Բացի կոմպիլյատորներից, գոյություն ունեն գրված ծրագիրը մեքենայական կոդի վերածող այլ ծրագրեր ևս, այսպես կոչված, **ինտերպրետատորներ**: Մրանք ծրագրի թարգմանությունն իրականացնում են հրաման առ հրաման՝ թարգմանված հրամանն ընթացքում իրականացնելով:

Առաջին աստիճանային լեզուները ստեղծվում էին կոնկրետ տիպի պրոցեսորների համար և հարմարեցվում էին դրանց առանձնահատկություններին. այդ պատճառով նման լեզուներն անվանեցին ծրագրավորման **ցածր մակարդակի** լեզուներ: Այստեղ «ցածր» բառը նշանակում է, որ տվյալ լեզվի օպերատորները հնարավորինս մոտ են մեքենայական կոդին: Ընդ որում՝ ծրագրավորման ցածր մակարդակի լեզվով գրված ծրագրերն, ըստ էության, շատ կուռ կառուցվածք ունեն և կատարման առումով ավելի քիչ ժամանակ են պահանջում:

50-ական թվականների երկրորդ կեսից սկսեցին ստեղծվել ծրագրավորման, այսպես կոչված, **բարձր մակարդակի** լեզուներ: Սրանք բնական լեզուներին հարմարեցված լեզուներ են և մարդուն առավել հասկանալի, քան համակարգչին: Ի տարբերություն ցածր մակարդակի լեզուների, այս լեզուներն արդեն կախված չեն համակարգչի տիպից:

Հետագայում տարբեր տիպի խնդիրներ լուծելու նպատակով բարձր մակարդակի զանազան լեզուներ ստեղծվեցին՝ *FORTRAN*, *COBOL*, *BASIC* և այլն: Ծրագրավորման *FORTRAN* (*FOR*mula *TRAN*slator) լեզուն գիտատեխնիկական հաշվարկներ կատարելու համար է, իսկ *COBOL*-ը (*CO*mmon *B*usiness - *O*riented *L*anguage) կոմերցիային առնչվող խնդիրներ լուծելու համար: Մինչդեռ *BASIC* (*B*eginner's *A*ll - *P*urpose *S*ymbolic *I*nstruction *C*ode) լեզուն աչքի էր ընկնում ուսուցման պարզությամբ և ուղղված էր սկսնակ ծրագրավորողներին:

80-ական թվականների սկզբին ծրագրավորման մեջ նոր որակ ապահովող **ալգորիթմական լեզուներ** մշակվեցին, որոնք թույլատրեցին անցում կատարել, այսպես կոչված, **կառուցվածքային ծրագրավորման**: Այս լեզուների հիմնական առանձնահատկությունն այն է, որ լեզվական նոր միջոցների շնորհիվ ալգորիթմական կառույցներն արդյունավետորեն ծրագրավորելու հնարավորություն ստեղծվեց:

Կառուցվածքային ծրագրավորման կայացման գործում մեծ դեր ունի ինչպես **Pascal (Պասկալ)** լեզուն, որը մեր դասընթացի մասն է կազմելու, այնպես էլ լայնորեն հայտնի **C(Մի)** լեզուն, որը թույլատրում է արագագործ ծրագրեր ստեղծել: *Պասկալ* ալգորիթմական լեզվի հիման վրա ստեղծվեցին *Object Pascal*, իսկ *QBasic* լեզվի հիման վրա՝ *Visual Basic* օբյեկտային կողմնորոշմամբ լեզուները:

C լեզվի հիման վրա 1980 թվականին ստեղծվեց **C++ օբյեկտային կողմնորոշմամբ** ծրագրավորման լեզուն: Մեր դասընթացի երկրորդ մասը նվիրված է այս լեզվին: Օբյեկտային կողմնորոշմամբ ծրագրավորման լեզուների առանձնահատկությունն այն է, որ հիմնված են **սվյալները** և դրանք մշակող **մեթոդները** միավորող **ծրագրային օբյեկտների** վրա:

20-րդ դարի 90-ական թվականներին, **Համացանցի** զարգացմանը զուգընթաց, ծրագրավորման այնպիսի նոր լեզուներ ստեղծվեցին, որոնք թույլատրում են տարբեր օպերացիոն հենքերի վրա աշխատող ծրագրեր ստեղծել: Այսինքն՝ մինևույն ծրագիրը կարող է աշխատել **Համացանցին** միացված ցանկացած համակարգչի վրա՝ անկախ դրա ընթացիկ օպերացիոն համակարգից (*WINDOWS*, *LINUX*, *Mac OS* և այլն): Նման լեզուներից է օբյեկտային կողմնորոշմամբ **Java** (Ջավա) լեզուն, որը ստեղծվել է C++-ի հիմքում: *Java* լեզուն ներկայումս աշխարհում լայնորեն կիրառվող երկրորդ լեզուն է՝ *Basic*-ից հետո:

1. ԾՐԱԳՐԱՎՈՐՄԱՆ ՊԱՍԿԱԼ ԼԵԶՎԻ ՀԻՄՈՒՆՔՆԵՐԸ



§ 1.1

ԾՐԱԳՐԱՎՈՐՄԱՆ ՊԱՍԿԱԼ ԼԵԶՈՒ: ԼԵԶՎԻ ԱՇԽԱՏԱՆՔԱՅԻՆ ՄԻՋՎԱՅՐԸ

Ծրագրավորման **Պասկալ** լեզուն ստեղծվել է 1968-1971 թվականներին՝ շվեյցարացի գիտնական **Նիկլաուս Վիրտի** կողմից, և անվանվել ֆրանսիացի նշանավոր մաթեմատիկոս, փիլիսոփա **Բլեզ Պասկալի** անունով: Ի սկզբանե լեզուն ստեղծվել է *ուսուցողական* նպատակով՝ որպեսզի հնարավորինս մատչելի դառնա ծրագրավորման հիմունքներ ուսուցանելը: Սակայն ստեղծված լեզուն պարզ *քերականություն*, կուռ *կառուցվածքայնություն*, կիրառման ոլորտների բազմազանության շնորհիվ շուտով դարձավ ժամանակի ամենագործածական լեզուներից մեկը: Պասկալը *կառուցվածքային ծրագրավորման* առաջին լեզուներից է. այստեղ կառուցվածքային ծրագրավորումը լիարժեքորեն արտահայտվել է ոչ միայն ծրագրի տարբեր բաղկացուցիչ մասերի միջև կապերի կարգավորման, այլև *սիմվոլների կառուցվածքայնացման* շնորհիվ:

Ներկայացնենք **Պասկալ** լեզվի հիմնական առանձնահատկություններից մի քանիսը.

- *Լեզվի պարզ քերականությունը:* Հիմնական հասկացությունների փոքրաթիվությունը: **Պասկալ**-ով գրված ծրագրի ընթեռնելիության պարզությունը:
- **Պասկալ**-ի կոմպիլյատորի և այդ լեզվով գրված ծրագրի կողմից *հաշվիչ համակարգին ներկայացվող բավական պարզ պահանջները:*
- Լեզվի ունիվերսալությունը. **Պասկալ** լեզուն գործնականում կիրառելի է ծրագրավորման ցանկացած տիպի խնդիրների համար (հաշվարկային, տնտեսագիտական և այլն):
- *Կառուցվածքային ծրագրավորման լեզու է*, որտեղ ընտրվել է ծրագրավորման **վերից վար** սկզբունքը, իսկ լեզվի հետագա տարբերակները (*Turbo Pascal 7.0*, *Borland Pascal*) նաև օբյեկտային կողմնորոշմամբ են:

Մենք ուսումնասիրելու ենք լեզվի **Տուրբո Պասկալ** (հետագայում՝ **Պասկալ**) տարբերակը, որը մշակվել է 1992 թվականին՝ *Borland International* ֆիրմայի կողմից: Այս տարբերակի կոմպիլյատորը (ինչպես բոլոր *Turbo* կոմպիլյատորները) բավական արագագործ է: Ընդ որում՝ լեզուն օժտված է նաև աշխատանքային հարուստ *ինտերգրացված (ամբողջական) միջավայրով*:

Միջավայրը թույլատրում է.

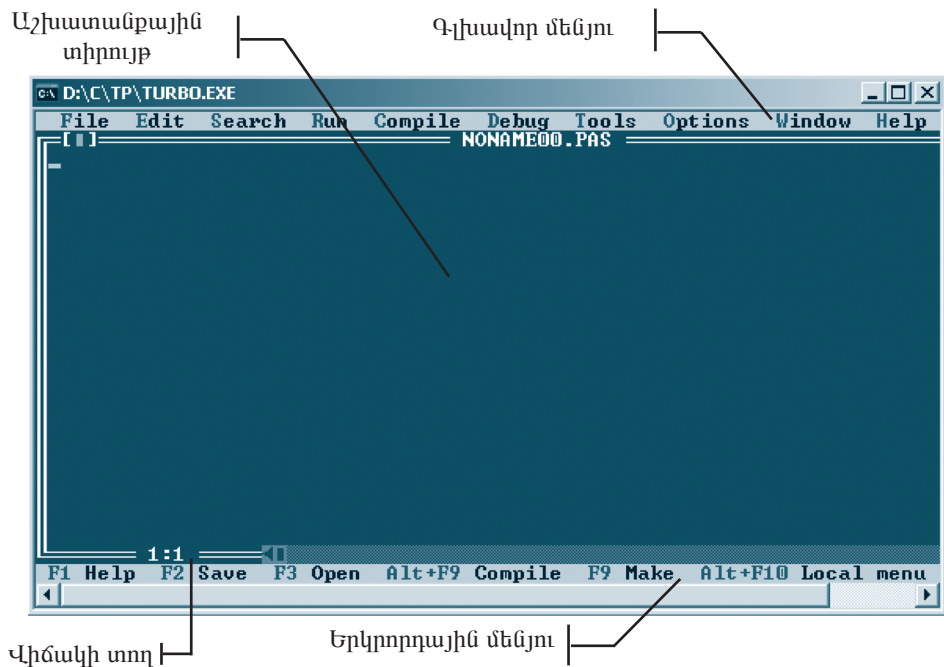
- ստեղծել ծրագրի տեքստը,
- իրագործել ծրագրի կոմպիլյացիան,
- ուղղել առկա քերականական սխալները,

- ծրագրի առանձին բաղկացուցիչ մասերից (ներառյալ նախապես մշակված ստանդարտ ծրագրային մոդուլները) միասնական աշխատող ծրագիր ստեղծել,
- կարգաբերել ծրագիրն ու իրագործել այն,
- օգտվել օգնության ծավալուն տեղեկատվական համակարգից:

Պասկալի *աշխատանքային միջավայրն* ակտիվացնելու նպատակով անհրաժեշտ է.

- Պասկալի ծրագրային փաթեթը ներառող հիշող սարքի վրա (օրինակ՝ համակարգչի կոշտ սկավառակի վրա) գտնել *TP*, *TURBO*, *TURBOPAS* կամ *PASCAL* անվանումներից որևէ մեկը կրող թղթապանակն ու բացել այն,
- բացված թղթապանակում ընտրել *Turbo.exe* ֆայլն ու իրագործել այն (սեղմել *Enter* ստեղծը կամ դրա վրա մկնիկի ձախ սեղմակի կրկնակի սեղմում կատարել):

Այս քայլերի արդյունքում բացվում է լեզվի ինտեգրացված աշխատանքային միջավայրը ներկայացնող հետևյալ պատուհանը (նկ. 1.1):



Նկ. 1.1. Տուրբո Պասկալի աշխատանքային միջավայրը

Բացված պատուհանում ֆունկցիոնալ առումով առանձնացվում են հետևյալ տեղանասերը.

- գլխավոր մենյու,
- աշխատանքային տիրույթ,
- վիճակի տող,
- երկրորդային մենյու:

Գլխավոր մենյուն կարելի է ակտիվացնել երկու եղանակով՝ սեղմելով ստեղնաշարի վերին մասում առկա **F10 ֆունկցիոնալ սլեյդեր** կամ օգտվելով մկնիկի ձախ սեղմակից:

Գլխավոր մենյուն պարունակում է հետևյալ բաղադրիչները.

- **File** – ֆայլի հետ կապված հնարավոր գործողություններ է ընդգրկում (ստեղծել նոր ֆայլ (*New*), բացել նախկինում ստեղծվածը (*Open*), պահպանել աշխատանքային միջավայրում տվյալ պահին առկա ծրագիրը (*Save, Save as...*) և այլն):
- **Edit** – թույլատրում է աշխատանքային միջավայրում առկա ծրագրի տեքստը խմբագրել (տեքստի ընտրված հատվածը պատճենել (*Copy*), տեղադրել (*Paste*), ընտրված հատվածը ջնջել (*Delete*), խմբագրական վերջին գործողությունն անտեսել / վերականգնել (*Undo/Redo*) և այլն):
- **Search** – թույլատրում է ներմուծված նմուշի համաձայն ծրագրի տեքստում որոնում իրականացնել (*Search*) և, անհրաժեշտության դեպքում, այն փոխարինել (*Replase*) նոր տեքստով:
- **Run** – թույլատրում է հնարավոր մի քանի ռեժիմներով իրագործել գրված ծրագիրը:
- **Compile** – ծրագիրը կոմպիլյացիայի (թարգմանման) ենթարկելու մի քանի եղանակ է պարունակում:
- **Debug** – հնարավորություն է տալիս ծրագրում առկա տրամաբանական սխալները գտնել:
- **Tools** – որոշ լրացուցիչ միջոցներ է պարունակում:
- **Options** – թույլատրում է սահմանել կոմպիլյացիայի և ինտեգրացված միջավայրի նախագծման համար անհրաժեշտ պարամետրերը:
- **Window** – պարունակում է պատուհանների հետ կապված հիմնական գործողությունները (բացել, փակել, տեղաշարժել, չափերը փոփոխել):
- **Help** – օգնության համակարգ է տրամադրում:

Մենյուի բաղադրիչները կարելի է ակտիվացնել նաև *Alt* ստեղնի և անհրաժեշտ բաղադրիչի անվան մեջ առկա կարմիր գույնի պայմանանիշի միաժամանակյա սեղմումով (օրինակ՝ *ALT-F*-ը բացում է *File* մենյուն):

Ինտեգրացված միջավայրի **աշխատանքային փիրույթում** կարելի է տարբեր պատուհաններ բացել՝ **խմբագրվող տեքստի, օգնության, կարգաբերման** և այլն:

Պատուհանի **վերնագրային** մասում գրվում է ծրագրի տեքստը պարունակող ֆայլի անվանումը:

Վիճակի փողը ցույց է տալիս տեքստային կուրսորի գտնվելու տողի և սյան համարները:

Երկրորդային մենյուն որոշ կարևոր գործողություններ և դրանց համապատասխանող ստեղների համադրություններ է պարունակում:

Միջավայրի խմբագիրը ծրագրի տեքստը ներմուծելու և խմբագրելու համար անհրաժեշտ միջոցներ է պարունակում: Ընդ որում՝ խմբագրման ռեժիմն ավտոմատ ակտիվացվում է **Պասկայի** ինտեգրացված միջավայր մտնելուն պես:

Գիտարկենք միջավայրի տեքստային ռեժիմի աշխատանքի հիմնական սկզբունքները: Ծրագրի տեքստը ներմուծվում է ստեղնաշարից, ընդ որում՝ խմբագրիչի պատուհանին անընդհատ թարթող կուրսորը ցույց է տալիս ներմուծվող ընթացիկ պայմանանշանի դիրքը: Յուրաքանչյուր տող ներմուծելուց հետո հաջորդ տողին կարելի է անցնել *ENTER*-ի միջոցով: Յուրաքանչյուր տող կարող է ամենաշատը 126 պայմանանշան պարունակել:

Պասկալի ինտեգրացված միջավայրում խմբագրիչի պատուհանը, ըստ ընտրված ռեժիմի, կարող է 25 կամ 50 տող պարունակել:

Ներմուծված տեքստի վրայով հնարավոր է *տեղաշարժվել* հետևյալ ստեղների միջոցով.

Home – վերադարձ ընթացիկ տողի սկզբնամաս,

End – անցում ընթացիկ տողի վերջնամաս,

PgUp – ընթացիկ տողից մեկ էջի չափով (25 կամ 50 տող) անցում կատարել դեպի ծրագրի սկզբնամաս (վերև),

PgDn – ընթացիկ տողից մեկ էջի չափով անցում կատարել ներքև,

Ctrl-PgUp – վերադարձ ծրագրի սկիզբ,

Ctrl-PgDn – անցում դեպի ծրագրի տեքստի վերջ:

Ստեղնաշարի ←, ↑, →, ↓ ստեղների միջոցով կուրսորը մեկ պայմանանշանի չափով տեղաշարժվում է համապատասխանաբար դեպի ձախ, վերև, աջ կամ ներքև:

Կուրսորի ընթացիկ դիրքից անմիջապես ձախ ընկած պայմանանշանը կարելի է *զնջել* *Backspace*, իսկ ընթացիկ պայմանանշանը՝ *Delete* ստեղնի օգնությամբ:

Սխալմամբ ջնջված ինֆորմացիան քայլ առ քայլ կարելի է կրկին *վերսկսնվել* *Alt-Backspace* ստեղների համատեղ հաջորդական սեղմումներով:

Խմբագրիչի միջավայրում կարելի է աշխատել նաև ծրագրի նախապես նշված հատվածի հետ. ծրագրի որևէ հատված կարելի է *նշել* *Shift* և ←, ↑, →, ↓, *PgDn*, *PgUp* ստեղներից անհրաժեշտը միաժամանակ սեղմելու միջոցով, կամ մկնիկի ձախ սեղմակը սեղմած վիճակում այն անհրաժեշտ ուղղությամբ տեղաշարժելով:

Նշված հատվածը հնարավոր է.

Ctrl-ky – ջնջել,

Ctrl-kc – պատճենել,

Ctrl-kv – տեղափոխել (նախկին տեղից այն ջնջելով),

Ctrl-kw – գրանցել ֆայլի մեջ,

Ctrl-kr – ֆայլից կարդալ,

Ctrl-kp – տպագրել:

Ծրագրի տեքստը ներմուծելուց հետո այն կարելի է *թարգմանել* և *իրագործել* օգտվելով *Ctrl-F9* ստեղներից: Այս ստեղների համատեղ սեղմման արդյունքում ծրագիրը կենթարկվի *կոմպիլյացիայի*. Եթե այն քերականորեն ճիշտ է, ապա դրանից հետո կենթարկվի, այսպես կոչված, *կոմպանովկայի (միաշուրում)*: Այս փուլում, եթե ծրագիրը *սպանդարտ ֆունկցիաների* կանչեր պարունակի, ապա այն կհամալրվի համապատասխան ֆունկցիաներով, իսկ վերջում արդեն ամբողջացած ծրագիրը կպահպանվի *.OBJ* ընդլայնմամբ ֆայլում: Այս ֆայլը թարգմանվում է *մեքենայական կոդի* և պահպանվում *EXE* ընդլայնմամբ ֆայլում, որն արդեն պատրաստ է իրագործ-

ման. այժմ այն *բեռնավորվում է մեքենայի օպերատորիվ հիշողություն* և իրագործվում հրաման առ հրաման:

Ծրագրի աշխատանքից հետո էկրանին կրկին բերվում է խմբագրիչի պատուհանը:

ՕՉՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ **Տուրքո Պասկայի ինտեգրացված միջավայրում աշխատելիս կարելի է օգտվել հետևյալ հիմնական հրամաններից և դրանց հետ կապված արագագործ սրեղներից.**

Alt+F5 (*Alt* և *F5* ստեղների համատեղ սեղմում) – ծրագրի աշխատանքի արդյունքի դիտում,

F2 – աշխատանքային տիրություն առկա ծրագրի պահպանում ֆայլի մեջ,

F3 – նախկինում պահպանված ծրագրի ակտիվացում,

Alt-F3 – ակտիվ պատուհանի փակում,

Alt-X – ելք *Պասկայի* միջավայրից,

F1 – օգնության ներքին համակարգի կանչ,

Ctrl-F1 – կուրսորի միջոցով ընտրված հրամանի վերաբերյալ տեղեկության տրամադրում,

Ctrl-Y – ընթացիկ տողի ջնջում,

Ctrl-Insert – ընդգծված հատվածի պատճենում բուֆերի մեջ,

Shift-Insert – կուրսորի դիրքից սկսած բուֆերում պահպանված հատվածի տեղադրում:

- ◆ **Պասկայ լեզվի հիման վրա հետագայում սրեղծվել է *Delphi* լեզուն, որը *Windows* միջավայրում աշխատող վիզուալ (գիպոդական) ծրագրավորման համակարգ է:**



1. Ի՞նչ նպատակով է նախագծվել Պասկայ լեզուն:
2. Թվարկեք Պասկայ լեզվի չեզ հայտնի առանձնահատկությունները:
3. Ի՞նչ հնարավորություններ է ընձեռում Պասկայի աշխատանքային միջավայրը:
4. Ի՞նչ ֆունկցիոնալ բաղադրիչներ է պարունակում ինտեգրացված միջավայրը:
5. Թվարկեք ինտեգրացված միջավայրի գլխավոր մենյուի չեզ հայտնի բաղկացուցիչ մասերը:
6. Ինչի՞ համար է միջավայրի խմբագրիչը:
7. Ծրագրի տեքստով տեղաշարժվելու ի՞նչ միջոցներ գիտեք:
8. Ինչպե՞ս կարելի է վերականգնել սխալմամբ ջնջված վերջին ինֆորմացիան:
9. Ինչպե՞ս կարելի է նշված հատվածը՝
 - ա) տեղափոխել,
 - բ) ջնջել:
10. Ի՞նչ փուլեր է անցնում ծրագիրը *Ctrl* և *F9* սրեղների համատեղ սեղմումով:

§ 1.2 ՊԱՏԿԱԼ ԼԵԶՎԻ ՏԱՐԲԵՐԸ

Ծրագրավորման Պատկալ լեզվի *այբուբենը* բաղկացած է *տառերից, քվանշաններից, հայտուկ պայմանանշաններից* և *առանցքային բառերից*:

Լեզվում կիրառում են *լատինական այբուբենի մեծատառերն* ու *փոքրատառերը*.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

Նշենք, որ Պատկալ լեզուն տարբերություն չի դնում մեծատառերի և փոքրատառերի միջև, օրինակ՝ լեզվի *կոնսիլյայտորի* կողմից *VAR* և *var* բառերն ընկալվում են միևնույն կերպ:

Որպես քվանշաններ լեզվում օգտագործվում են արաբական *0, 1, 2, ..., 9* քվանշանները:

Լեզվում կիրառվում են հետևյալ *հայտուկ պայմանանշանները*.

{ } [] () ‘ ; , . = + - * / : _ = > < # & @

Սրանք, կախված կիրառումից, կարող են տարբեր իմաստներ ունենալ.

Լեզվում օգտագործում են նաև որոշակի *պայմանանշանների համադրություններ*, որոնք ընդունվում են որպես ամբողջություն, գրվում են կից՝ առանց բացատարանիչի և որոշակի իմաստ ունեն.

(* *) := >= <= <> ..

Բերված պայմանանշանների և դրանց համադրությունների նշանակությանը կձանոթանանք դրանք կիրառելու ընթացքում:

Պատկալում առկա *առանցքային բառերը* հայտուկ տերմիններ են, որոնք կիրառական որոշակի իմաստ ունեն, չեն կարող վերահիմաստավորվել և կիրառվել այլ կերպ, քան ընդունված է:

Լեզվում կիրառվում են հետևյալ առանցքային բառերը.

<i>and</i>	<i>downto</i>	<i>inline</i>	<i>procedure</i>	<i>type</i>
<i>array</i>	<i>else</i>	<i>interface</i>	<i>program</i>	<i>unit</i>
<i>asm</i>	<i>end</i>	<i>label</i>	<i>record</i>	<i>until</i>
<i>begin</i>	<i>file</i>	<i>mod</i>	<i>repeat</i>	<i>uses</i>
<i>case</i>	<i>for</i>	<i>nil</i>	<i>set</i>	<i>var</i>
<i>const</i>	<i>function</i>	<i>not</i>	<i>shl</i>	<i>while</i>
<i>constructor</i>	<i>goto</i>	<i>object</i>	<i>shr</i>	<i>with</i>
<i>destructor</i>	<i>if</i>	<i>of</i>	<i>string</i>	<i>xor</i>
<i>div</i>	<i>implementation</i>	<i>or</i>	<i>then</i>	
<i>do</i>	<i>in</i>	<i>packed</i>	<i>to</i>	

Այս դասընթացի շրջանակում աշխատելու ենք վերը բերված առանցքային բառերի մեծ մասի հետ:

Ծրագիր կազմելիս հաճախ է անհրաժեշտ լինում կիրառվող մեծություններին որոշակի *անվանումներ* տալ և հետագայում դրանց դիմել այդ անվանումներով: Տրված անվանումները կոչվում են **իդենտիֆիկատորներ (նույնացուցիչ)**: Սրանք *Պասկալում* կազմվում են լեզվի այբուբենի *տառերից* ու *թվերից*, իսկ հատուկ պայմանանշաններից կարող են պարունակել միայն *ընդգծման* () նշանը:

Իդենտիֆիկատորը պետք է սկսվի տառով և չի կարող առանցքային բառ լինել:

Հետևյալ գրառումները ճիշտ իդենտիֆիկատորներ են. $A2B$, DDd , A_3 :

Քանի որ լեզուն մեծատառերի և փոքրատառերի միջև տարբերություն չի դնում, ապա $BETA$, $Beta$ և $beta$ իդենտիֆիկատորները համարժեք են և նշում են միևնույն մեծությունը:

Եթե իդենտիֆիկատորը բաղկացած է մի քանի բառերից, ապա խորհուրդ է տրվում բաղադրիչ բառերը սկսել մեծատառերով կամ դրանց միջև ընդգծման նշան դնել: Օրինակ, $AlfaBeta$ կամ $ALFA_BETA$:

Այն մեծությունները, որոնց արժեքները ծրագրի կապարման ընթացքում չեն փոփոխվում, կոչվում են հասարարուն մեծություններ կամ հասարարուններ:

Հասարարունները ծրագրում կարող են ներկայացվել ինչպես կոնկրետ արժեքների, այնպես էլ անվանումների (իդենտիֆիկատորների) միջոցով: Ծանոթանանք *Պասկալում* կիրառվող երկու՝ թվային և սիմվոլային հաստատումների հետ:

Թվային հասարարունները գրվում են մաթեմատիկայից ձեռք ծանոթ գրելաձևով, ընդ որում՝ դրական թվի դեպքում $+$ նշանը կարող է բացակայել:

Պասկալում թվերը կարող են ներկայացվել **ամբողջ** և **իրական** տեսքերով: Իրական թվում ամբողջ և կոտորակային մասերն իրարից անջատվում են **կետի** ($.$) միջոցով: Իրական թիվը չի կարող սկսվել կամ ավարտվել կետով:

Պասկալի կանոններով՝ ճիշտ գրված հաստատուն թվային մեծություններ են, օրինակ, 1900 , $+5$, -7 , 2.85 , 0.02 , -10.802 , -6.0 թվերը:

Իրական թվերը ներկայացվում են մաս մեկ այլ՝ *ցուցային եղանակով*, որտեղ թվի 10-ական կարգը գրվում է E կամ e տառից հետո: Օրինակ՝ 0.003 կամ $0.3 \cdot 10^{-2}$ թիվը *Պասկալում* կարելի է գրել $0.3E-2$ կամ $0.03E-1$ տեսքով:

Սիմվոլային հասարարունը ստեղծանշարին առկա ցանկացած տառ, պայմանանշան կամ թվանշան է՝ առնված ապաթարցերի մեջ: Օրինակ՝

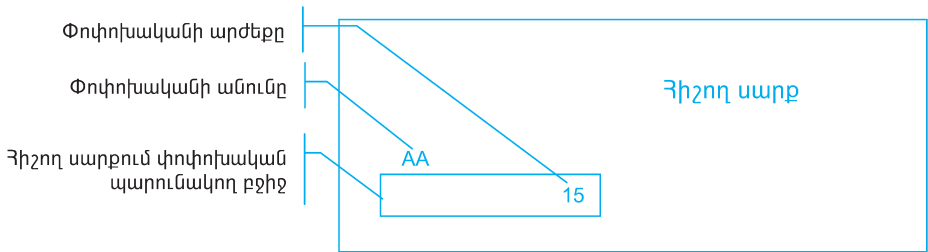
'c' – c տառն է, իսկ

)' – ը փակող փակագծի համար նախատեսված պայմանանշանը:

Ծրագրավորման մեջ կարևորագույն հասկացություններից մեկը **փոփոխականի** հասկացությունն է: Ցանկացած ծրագրում հետագա մշակման նպատակով անհրաժեշտ է լինում պահպանել թե՛ ներմուծվող տվյալները և թե՛ ծրագրի կատարման ընթացքում ստացվող տվյալները: Այդ նպատակով հիմնականում օգտվում են փոփոխականներից:

Այն մեծությունները, որոնց արժեքները ծրագրի կատարման ընթացքում կարող են փոփոխվել, կոչվում են փոփոխականներ:

Փոփոխականները ծրագրում ներկայացվում են *իդենտիֆիկատորների* միջոցով: Յուրաքանչյուր փոփոխականի անունը եզակի է և ծրագրի կատարման ընթացքում չի կարող փոփոխվել: Ընդ որում՝ համակարգչի մեջ յուրաքանչյուր փոփոխականի համար որոշակի ծավալով հիշողություն է հատկացվում (նկ. 1.2):



Նկ. 1.2. Փոփոխականի տեղակայումը համակարգչի հիշող սարքում

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

♦ **Իդենտիֆիկատորը կարող է ցանկացած երկարություն ունենալ, սակայն Պասկալ լեզվի համար խնաստալից են միայն դրա առաջին 63 պայմանականանները:**



1. Ո՞ր լեզվի այբուբենն է կիրառվում Պասկալ լեզվում:
2. Ի՞նչ է իդենտիֆիկատորը:
3. Նշե՛ք, թե հետևյալ իդենտիֆիկատորներից որոնք են սխալ և ինչու.

ա) X – 4,	ե) Pars_Kar,
բ) eps7,	զ) 3h26,
գ) Alfa 1,	է) a&b:
դ) ParsKar,	
4. Ո՞ր մեծություններն են կոչվում հաստատուներ:
5. Ի՞նչ է սիմվոլային հաստատուներ:
6. Ո՞ր մեծություններն են կոչվում փոփոխական և ինչպե՞ս են դրանք ներկայացվում ծրագրում:

§ 1.3 ՊԱՍԿԱԼ ԾՐԱԳՐԻ ԿԱՌՈՒՑՎԱԾՔՆ ՈՒ ՀԻՄՆԱԿԱՆ ԲԱԺԻՆՆԵՐԸ

Յուրաքանչյուր ծրագրավորման լեզու *ծրագիր կազմելու* (գրելու, ստեղծելու) իր կանոններն ունի: Պասկալով գրված ծրագրի հիմնական կառուցվածքը կարելի է ընդհանուր կերպով ներկայացնել հետևյալ կերպ.

ծրագրի վերնագիր,

ծրագրում կիրառվող մեծությունների հայտարարում,

ծրագրի հիմնական մարմին:

Ծրագրի վերնագիրը սկսվում է **PROGRAM** առանցքային բառով, որին հաջորդող մեկ կամ մի քանի բացատանիշից հետո գրվում է **ծրագրի անունը**. վերջինս *Պասկալի* կանոններին համապատասխանող ցանկացած իդենտիֆիկատոր է: Ծրագրի վերնագիրն ավարտվում է **կեղտ-ստորակետով (;)**:

Օրինակ՝ `PROGRAM XNDIR;`

Ընդհանրապես կարելի է ծրագրին վերնագիր չտալ, այն բաց թողնել: Սակայն անվան առկայությունն օգնում է ծրագիրը ճանաչելի, ընթեռնելի դարձնել, եթե այն արտահայտում է տվյալ աշխատանքի նպատակը: Օրինակ՝ ծրագրի վերնագիրը կարող է լինել

`PROGRAM ERANKYAN_MAKERES;`

`PROGRAM GAME_TETRIS;`

և այլն:

Պասկալի ցանկացած հրաման (օպերատոր) ավարտվում է կետ-ստորակետով:

Պասկալի կարևոր առանձնահատկություններից մեկն այն է, որ ծրագրում կիրառվող մեծությունները պետք է **նախօրոք նկարագրվեն**, կամ, որ նույնն է, **հայտարարվեն**: *Նկարագրությունների* կամ *հայտարարությունների* բաժինը կարող է ներառել հետևյալ հնարավոր բաղադրիչները.

նոր տիպերի նկարագրություններ,

հասարարությունների նկարագրություններ,

նշիչների նկարագրություններ,

փոփոխականների նկարագրություններ:

Նոր տիպերի նկարագրությունները սկսվում են **TYPE** առանցքային բառով, որին հաջորդում են ստեղծվող *նոր տիպերի* հայտարարությունները՝ իրարից ;-երով փոխանցատված: Նոր տիպերը ստեղծվում են *Պասկալում* նախասահմանված տիպերի հիման վրա: Օրինակ՝

`TYPE NOR_TIP=REAL;`

`TARIQ=BYTE;` և այլն,

որտեղ `NOR_TIP` և `TARIQ` իդենտիֆիկատորները ստեղծվող տիպերի անուններն են,

իսկ հավասարման (=) նշանին հաջորդում են *Պասկալում* նախասահմանված *REAL* և *BYTE* տիպերը (սրանց կծանոթանանք հաջորդ պարագրաֆում):

Հաստատումների հայտարարությունները սկսվում են **CONST** առանցքային բառով: Օրինակ՝

```
CONST e=2.7;
      tar= 'a'; և այլն:
```

Ըստ այս հայտարարման՝ *e* և *tar* իդենտիֆիկատորները ծրագրի կատարման ընթացքում արժեքները փոփոխել չեն կարող՝ *e*-ի արժեքը կմնա 2.7, իսկ *tar*-ի արժեքը՝ 'a' պայմանանշանը:

Նշիչների հայտարարությունը տրվում է առանցքային **LABEL** բառի օգնությամբ: Օրինակ՝

```
LABEL cikl, ab, 57;
```

Այստեղ **LABEL** բառից հետո, իրարից ստորակետերով անջատված, թվարկվել են *cikl*, *ab* և *57* նշիչները:

Նշիչ կարող է լինել ինչպես իդենտիֆիկատորը, այնպես էլ դրական ամբողջ թիվը:

Նշիչները ստեղծվում են անհրաժեշտության դեպքում՝ նշելու (անվանելու) համար այն հրամանները (օպերատորները), որոնց պետք է անցում կատարել՝ այսպիսով խախտելով ծրագրի կատարման հաջորդական ընթացքը: Նշիչը չի կարող մեկից ավելի օպերատորներ նշել, և հայտարարված նշիչը պետք է ծրագրում օգտագործել:

Օպերատորը նշելու համար անհրաժեշտ է նշիչի և համապատասխան օպերատորի միջև վերջակետ (:) դնել: Օրինակ՝ *Cikl : i = 1;*

Ծրագրում օգտագործված **փոփոխականները** պետք է **նախօրոք հայտարարել VAR** առանցքային բառի տակ: Օրինակ՝

```
VAR c:CHAR;
    x,k:BYTE;
    d,m,l:REAL;
```

Ինչպես երևում է օրինակից՝ նույն տիպի փոփոխականներ հայտարարելիս կարելի է դրանք խմբավորել ըստ միևնույն տիպի պատկանելության՝ իրարից ստորակետերով անջատելով: Իդենտիֆիկատորն ու տիպ արտահայտող առանցքային բառն իրարից պետք է փոխանջատել : -ով՝ վերջակետով:

Ծրագրի հիմնական մարմինը (իրագործվող հրամանների հաջորդականությունը) առնվում է **BEGIN** և **END** առանցքային բառերի մեջ:

BEGIN և END բառերի միջև ներառված օպերատորների համախումբն անվանում են բլոկ:

Ծրագրում կարող են բազմաթիվ բլոկներ ներառվել, սակայն միայն ծրագրի հիմնական մարմինը ներառող բլոկի **END** - ն է ավարտվում կետով:

Բլոկները կարող են լինել նաև *ներդրված* մեկը մյուսի մեջ *ներառված*:

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ **Թվով արտահայտված նշիչը կարող է [0;9999] միջակայքի որևէ թիվ լինել:**
- ◆ **Միևնույն օպերատորը կարող է մի քանի նշիչներ ունենալ, այս դեպքում դրանք իրարից պեպք է անջատել :- երով:**
- ◆ **Եթե ծրագիրը պարունակում է ոչ սրանդարպ, ծրագրավորողի կողմից ստեղծված ենթածրագրեր (ֆունկցիաներ, պրոցեդուրաներ), ապա դրանք կս պեպք է հայտարարել ծրագրի հիմնական մարմնից դուրս, դրանից առաջ:**



1. **Թվարկեք ծրագրի կառուցվածքի ձեզ հայտնի բաղադրիչները:**
2. **Որո՞նք են ծրագրի ճիշտ վերնագրեր.**
 - ա) Program L_9;
 - բ) PROGRAM kk+5;
 - գ) ProGram DD
 - դ) Program 5AB;
3. **Ո՞ր պնդումն է ճիշտ.**
CONST a=7; հայտարարությունը՝
 - ա) փոփոխականի հայտարարություն է,
 - բ) հաստատունի հայտարարություն է:
4. **Նշիչի n՞ր հայտարարություններն են ճիշտ.**
 - ա) LABEL a;
 - բ) label -5;
 - գ) LABEL a6;
 - դ) LABEL 1.25;
 - ե) label LL,67,k_8;
5. **Որո՞նք են փոփոխականների ճիշտ հայտարարություններ.**
 - ա) VAR a:b:REAL;
 - բ) VAR c:CHAR, d:REAL
 - գ) VAR 2:BYTE;
 - դ) VAR c,d:INTEGER;
 - ե) VAR k:REAL; B:BYTE;
6. **Ի՞նչ է բլոկը: Ո՞ր բլոկն է ավարտվում կետով (.):**

§ 1.4 ՏՎՅԱԼՆԵՐԻ ՊԱՐԶԱԳՈՒՅՆ ՏԻՊԵՐ: ՏՎՅԱԼՆԵՐԻ ԳՐԱՆՑՄԱՆ ԱՌԱՆՁՆԱՀԱՏԿՈՒԹՅՈՒՆՆԵՐԸ

Համակարգչում ներկայացվող ցանկացած մեծություն բնութագրվում է իր *փիպով*: Ծրագրում կիրառվող փոփոխականի տիպը կանխորոշվում է այն տվյալների տիպով, որ պետք է պարունակի:

Պասկալ լեզվում տվյալների *պարզ փիպեր* են համարվում *կարգային* և *իրական* տիպերը: Կարգային տիպի փոփոխականները կարող են վերջավոր քանակությամբ արժեքներ ունենալ, որոնք որոշակի կերպով կարգավորելով՝ կարելի է համարակալել: Այսպիսով՝ նման արժեքներից յուրաքանչյուրին կարելի է որոշակի ամբողջ թվով արտահայտված կարգահամար ամրագրել:

Կարգային տիպերից կժանոթանանք *ամբողջ*, *պրամաքանական*, *սիմվոլային*, *թվարկելի* ու *միջակայքային* տիպերին:

Պասկալում հնարավորություն ունենք աշխատելու մի քանի *ամբողջ փիպի* մեծությունների հետ, որոնք միմյանցից տարբերվում են հնարավոր արժեքների թույլատրելի միջակայքերով և տրամադրվող հիշողության ծավալներով (աղյուսակ 1.1):

Աղյուսակ 1.1

Ամբողջ տիպ		
Անվանումը	Երկարությունը (բայթերով)	Արժեքների միջակայքը
<i>BYTE</i>	1	0 ÷ 255
<i>WORD</i>	2	0 ÷ 65535
<i>INTEGER</i>	2	-32 768 ÷ 32 767
<i>SHORTINT</i>	1	-128 ÷ 127
<i>LONGINT</i>	4	-2147483648 ÷ 2147483647

Տրամաքանական փիպի փոփոխականի հնարավոր արժեքները երկուսն են՝ *TRUE* (*ճիշտ*) և *FALSE* (*սխալ*): Ընդ որում՝ *FALSE*-ին համապատասխանեցված է 0 կարգահամարը, իսկ *TRUE*-ին՝ 1-ը: Տրամաքանական տիպին տրամադրվում է 1 բայթ: Այն հայտարարվում է *BOOLEAN* առանցքային բառով, օրինակ՝ *T:BOOLEAN*:

Սիմվոլային փիպի մեծություն է համակարգչում ներկայացվող ցանկացած պայմանանշան՝ տառ, թիվ և ցանկացած հատուկ պայմանանշան: Սիմվոլային տիպի փոփոխականը հայտարարվում է *CHAR* առանցքային բառով:

Օրինակ՝ *c:CHAR*;

Յուրաքանչյուր պայմանանշանի (սիմվոլի) համապատասխանեցվում է 0..255 միջակայքի որոշ ամբողջ թիվ, որը հանդիսանում է տվյալ պայմանանշանի մեքենայական ներքին կոդը՝ ըստ *ASCII սրանդարտի*:

Թվարկելի փիպը տրվում է իր հնարավոր արժեքների թվարկմամբ, որոնք վերցվում են ձևավոր փակագծերի { } մեջ և իրարից բաժանվում ստորակետերով: Ընդ

որում՝ թվարկելի տիպում կարող է ամենաշատը 65536 բաղադրիչ ներառվել, որտեղ առաջին բաղադրիչը համարակալվում է 0 թվով, հաջորդը՝ 1-ով և այլն: Թվարկվող արժեքներից յուրաքանչյուրը բնորոշվում է որևէ անվամբ՝ իդենտիֆիկատորով: Օրինակ՝

```
TYPE color= {RED,BLACK,WHITE};
var x:color;
```

Այստեղ x -ը նշված երեք գույներից որևէ ցանկացած արժեք ընդունող $color$ թվարկելի տիպի փոփոխական է:

Միջակայքային տիպը ցանկացած կարգային տիպի վրա հիմնված տիպ է, որը ստացվում է հիմնային հանդիսացող տվյալ կարգային տիպի մի մասից: Այս տիպը տրվում է իր ստորին ու վերին եզրային արժեքների միջոցով՝ դրանք իրարից բաժանելով երկու կետով (..): Օրինակ՝

```
Type t= 'a'..'d';
b= 1..10;
Var alpha: t;
number: b;
```

Ըստ այս հայտարարության՝ $alpha$ -ն կարող է ընդունել 'a', 'b', 'c', 'd' արժեքները, իսկ $number$ -ը՝ 1-ից 10 ամբողջ թվերից ցանկացածը:

Իրական տիպի թիվը համակարգչի հիշողությունում հաճախ գրառվում է որոշակի մոտավորությամբ, որը կախված է իրական թվի ներքին (մեքենայական) ներկայացման ձևաչափից: Պասկալը իրական թիվ ներկայացնելու հինգ տիպեր ունի՝ *REAL*, *SINGLE*, *DOUBLE*, *EXTENDED* և *COMP* (աղյուսակ 1.2): Ընդ որում՝ վերջին 4 տիպերը կարելի է կիրառել միայն կոմպիլյացիայի հատուկ ռեժիմի պայմաններում՝ այսպես կոչված, **թվաբանական համապրոցեսորի** կցմամբ: Այս դեպքում թվաբանական համապրոցեսորը հաշվարկային գործողությունները միշտ իրականացնում է *extended*-ի ձևաչափին համապատասխան, իսկ մնացած տիպերին համապատասխանող արժեքները ստացվում են արդյունքի վերջին մասի անտեսմամբ՝ հատմամբ:

Աղյուսակ 1.2

Իրական տիպ			
Տիպը	Բայթերի քանակը	Թվի բաղադրիչ թվանշանների քանակը	Արժեքների միջակայքը
<i>REAL</i>	6	11-12	$2.9 \cdot 10^{-39} \div 1.7 \cdot 10^{38}$
<i>SINGLE</i>	4	7-8	$1.5 \cdot 10^{-45} \div 3.4 \cdot 10^{38}$
<i>DOUBLE</i>	8	15-16	$5.0 \cdot 10^{-324} \div 1.7 \cdot 10^{308}$
<i>EXTENDED</i>	10	19-20	$3.4 \cdot 10^{-4932} \div 1.1 \cdot 10^{4932}$
<i>COMP</i>	8	19-20	$-2 \cdot 10^{63} + 1 \div 2 \cdot 10^{63} - 1$

COMP տիպն իրականում լայն միջակայք ապահովող **ամբողջ տիպ** է, որը մեկնաբանվում է որպես կոտորակային մաս չունեցող իրական տիպ:

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ **Համապրոցեսորը միացնելու համար պետք է ակտիվացնել մենյուի բողոքի Options ենթամենյուի Compiler հրամանն ու բացված Compiler Options վահանակից Numering Processing դաշտի 8087/80287 գրառման շախ մասում նշում կատարել:**
- ◆ **Համապրոցեսորի առկայության դեպքում խորհուրդ է փոխարեն կիրառել SINGLE կամ DOUBLE տիպերից որևէ մեկը, քանի որ REAL-ը հարմարեցված է առանց համապրոցեսորի աշխատելու ռեժիմին:**
- ◆ **COMP տիպը հիմնականում հարմար է կիրառել հաշվապահության մեջ՝ դրամական հաշվարկներ իրականացնելիս:**



1. Պասկալ լեզվում փոփոխության n° տիպերն են անվանում պարզ:
2. Թվարկեր կարգային տիպերը:
3. Ի՞նչ արժեքներ են ընդունում տրամաբանական տիպի փոփոխականները:
4. Ի՞նչ է սիմվոլային տիպը:
5. Ի՞նչ է թվարկելի տիպը:
6. Ո՞ր տիպերն են կոչվում միջակայքային:

ՍՏԱՆԳԱՐՏ ՄԱԹԵՄԱՏԻԿԱԿԱՆ ՖՈՒՆԿՑԻԱՆԵՐ

§ 1.5 ԵՎ ԱՐՏԱՀԱՅՏՈՒԹՅՈՒՆՆԵՐԻ ՊԱՏԱՀԱԿԱՆ ԹՎԵՐԻ ԳԵՆԵՐԱՑՈՒՄ

Պասկալ լեզվում կարգային և իրական տիպերի հետ աշխատող մի շարք **սպասնորարդ ֆունկցիաներ** կան: Սրանք նախօրոք մշակված և Պասկալին կցված ծրագրային միջոցներ են:

Ծանոթանանք կարգային տիպի հետ կիրառվող որոշ ստանդարտ ֆունկցիաների:

Եթե x արգումենտը կարգային տիպի է, ապա **ORD(x)**-ը վերադարձնում է x արգումենտի կարգային համարը: Ընդ որում՝ եթե x -ը ամբողջ տիպի է, ապա $ORD(x)=x$, տրամաբանական տիպի դեպքում **ORD** ֆունկցիայի արժեքը հավասար է արժեքի կարգահամարին՝ 0 կամ 1, սիմվոլային տիպի դեպքում՝ արժեքի ներքին կոդին, որը 0-ից 255 միջակայքի որևէ թիվ է, թվարկելի տիպի դեպքում՝ դիրքի համարին, որը 0-ից 65535 միջակայքի որևէ թիվ է, իսկ միջակայքային տիպի դեպքում արժեքը կախված է համապատասխան հիմնային տիպից:

Ծանոթանանք կարգային արգումենտի համար սահմանված ևս մի քանի ֆունկցիաների.

- **PRED(x)**-ը վերադարձնում է x -ին նախորդող արգումենտի արժեքը, որն ունի $ORD(x)-1$ կարգային համարը:
- **SUCC(x)**-ը վերադարձնում է x -ին հաջորդող արգումենտի արժեքը, որն ունի $ORD(x)+1$ կարգային համարը:

Օրինակ, եթե $x=5$, ապա $PRED(x)=4$, $SUCC(x)=6$,
 եթե $x='c'$, ապա $PRED(x)='b'$, $SUCC(x)='d'$:

x և y ամբողջ տիպի արգումենտների համար սահմանված են նաև հետևյալ ֆունկցիաները.

- DEC(x,y)** – x -ի արժեքը նվազեցվում է y -ի չափով, իսկ եթե y -ը բացակայում է՝ ($DEC(x)$), ապա x -ի արժեքը նվազում է 1 -ով:
- INC(x,y)** – x -ի արժեքն աճում է y -ի չափով, իսկ եթե y -ը բացակայում է՝ ($INC(x)$), ապա x -ի արժեքն աճում է 1 -ով:
- ODD(x)** – վերադարձնում է $TRUE$, եթե x -ը կենտ թիվ է, և $FALSE$ ՝ հակառակ դեպքում:

Պասկալում իրական տիպի արգումենտների համար նույնպես *ստանդարտ մաթեմատիկական ֆունկցիաներ* կան մշակված: Աղյուսակ 1.3-ում բերվել են դրանց համառոտ նկարագրությունները:

Աղյուսակ 1.3

Մաթ. ֆունկցիան	Համարժեք տեսքը Պասկալում	Արգումենտի տիպը	Արդյունքի տիպը	Գործողությունը
$\sin x$	$SIN(x)$	ամբողջ, իրական	իրական	ռադիաններով արտահայտված x արգումենտի սինուսը
$\cos x$	$COS(x)$	-	-	ռադիաններով արտահայտված x արգումենտի կոսինուսը
$\arctg x$	$ARCTAN(x)$	-	-	ռադիաններով արտահայտված x արգումենտի արկտանգենսը
$\ln x$	$LN(x)$	-	-	x արգումենտի բնական հիմքով լոգարիթմը
e^x	$EXP(x)$	-	-	e բնական թվի x աստիճանը
\sqrt{x}	$SQRT(x)$	-	-	քառակուսի արմատ x -ից
x^2	$SQR(x)$	ամբողջ, իրական	ամբողջ, իրական	x -ի քառակուսին
$ x $	$ABS(x)$	-	-	x -ի մոդուլը (բացարձակ արժեքը)
$[x]$	$INT(x)$	-	ամբողջ	x -ի ամբողջ մասը
$\{x\}$	$FRAC(x)$	-	իրական	x -ի կոտորակային մասը

Ծրագրավորման մեջ հաճախ է անհրաժեշտ լինում աշխատել պատահական կերպով արժեքավորվող մեծությունների հետ: *Պատկալում* պատահական արժեքներ վերարտադրելու նպատակով կիրառում են հետևյալ միջոցները.

RANDOMIZE – պատահական թվեր վերարտադրող համակարգի սկզբնարժեքավորում: Այս գործընթացը հիմնվում է համակարգչային ժամանակի ընթացիկ տվյալների վրա:

RANDOM – ֆունկցիա է, որը $[0;1)$ միջակայքից որևէ պատահական թիվ է վերադարձնում:

RANDOM(x) – վերադարձնում է $[0,x-1]$ միջակայքի որևէ պատահական ամբողջ թիվ. այստեղ x -ը ցանկացած ամբողջ տիպի դրական թիվ է:

ՕԳՏԱԿԱՐ Է ԻՄԱՆՍԸ

- ◆ **PRED ֆունկցիայի արժեքն անորոշ է, եթե կիրառվում է արգումենտի հնարավոր արժեքների ըստ աճման կարգավորված հաջորդականության առաջին արժեքի վրա:**
- ◆ **SUCC ֆունկցիայի արժեքն անորոշ է, եթե կիրառվում է արգումենտի հնարավոր արժեքների ըստ աճման կարգավորված հաջորդականության վերջին արժեքի վրա:**
- ◆ **π թիվը Պատկալ լեզվում սահմանված է որպես *PI* անվամբ հաստատում:**
- ◆ **Եռանկյունաչափական ֆունկցիաների արգումենտները պետք է չլինեն ռադիաններով: Աստիճանով արտահայտված անկյունը պետք է վերածել ռադիանի՝**

օգտվելով $X_{\text{ռադ.}} = \frac{X_{\text{աստ.}} \cdot \pi}{180}$ բանաձևից:

- ◆ **Բնական հիմքով լոգարիթմից բացի, այլ հիմքով լոգարիթմական ֆունկցիայի արժեքը հաշվելու համար անհրաժեշտ է օգտվել լոգարիթմի մի հիմքից մյուսին անցնելու $\log_a b = \frac{\log_c b}{\log_c a}$ բանաձևից՝ c հիմքի փոխարեն կիրառելով բնական e հիմքը: Այսպիսով՝ $\log_a b \rightarrow \ln(b)/\ln(a)$:**
- ◆ **Պատկալ լեզվում արգումենտի քառակուսի աստիճանից բարձր աստիճան հաշվող սրանդարտ ֆունկցիա չկա, այդ պատճառով դրական արգումենտի աստիճանը հաշվելու համար կարելի է օգտվել մաթեմատիկայից հայտնի $a^n = e^{n \ln(a)}$ առնչությունից:**



1. Ի՞նչ արժեքներ կընդունի $ORD(x)$ ֆունկցիան, եթե x -ը հավասար է.
 - ա) $TRUE$,
 - բ) 7,
 - գ) 'a':

2. Ի՞նչ արժեքներ կընդունի $SUCC(x)$ ֆունկցիան, եթե x -ը հավասար է.
 - ա) $FALSE$,
 - բ) 10,
 - գ) -7,
 - դ) 'a':

3. Ի՞նչ արժեքներ կընդունի $PRED(x)$ ֆունկցիան, եթե x -ը հավասար է.
 - ա) $TRUE$,
 - բ) -8,
 - գ) 100,
 - դ) 'y':

4. Եթե $x=7$ և $y=5$, ապա ի՞նչ արժեք կընդունի x -ը $DEC(x,y)$ -ի արդյունքում.
 - ա) 2,
 - բ) 12:

5. Եթե $a=7$ և $b=2$, ապա ի՞նչ արժեք կընդունի a -ն $INC(a,b)$ -ի արդյունքում.
 - ա) 10,
 - բ) 14,
 - գ) 9,
 - դ) որոշված չէ:

6. Եթե $a=13$, ապա n ՞րն է ճիշտ.
 - ա) $ODD(a)=TRUE$,
 - բ) $ODD(a)=FALSE$:

7. Ինչպե՞ս կարելի է $[0;9]$ միջակայքի որևէ պատահական թիվ սրանալ:
8. Ինչպե՞ս սրանալ $[0;1)$ միջակայքի պատահական թիվ:
9. Ինչպե՞ս սրանալ $[10;19]$ միջակայքի պատահական թիվ:

§ 1.6

ԹՎԱԲԱՆԱԿԱՆ ԵՎ ՏՐԱՄԱԲԱՆԱԿԱՆ ԱՐՏԱՀԱՅՏՈՒԹՅՈՒՆՆԵՐ

Թվաբանական արտահայտությունները կարող են պարունակել փակագծերի և գործողությունների նշանների միջոցով իրար համակցված հաստատումներ, փոփոխականներ, ստանդարտ ֆունկցիաներ: Կարելի է ասել, որ արտահայտությունները նոր արժեքներ ստանալու միջոցներ են հանդիսանում: Արտահայտությունը, մասնավորապես, կարող է բաղկացած լինել միայն մեկ բաղադրիչից՝ հաստատումից, փոփոխականից կամ ստանդարտ ֆունկցիայի կանչից. այս դեպքում արտահայտության արժեքի տիպը համընկնում է տվյալ բաղադրիչի տիպի հետ: Ընդհանուր դեպքում՝

արտահայտության արժեքի տիպը որոշվում է դրա մեջ ներառված արգումենտների տիպերից և դրանց նկատմամբ կիրառված գործողություններից:

Պասկալում որոշված են հետևյալ **գործողությունները**.

ուևար (մեկտեղանի)՝ +, -, NOT, @,
մուլտիպլիկատիվ՝ *, /, DIV, MOD, AND, SHL, SHR,
ադիտիվ (երկտեղանի)՝ +, -, OR, XOR,
հարաբերման գործողություններ՝ =, <, >, >=, <=, <>, IN:

Ուևար + և - գործողությունները կիրառվում են արգումենտի նշանի (դրական, բացասական) իմաստով, ընդ որում՝ դրական արգումենտի նշանը կարելի է չտալ:

Արտահայտության արժեքը հաշվելիս դրանում կիրառված գործողությունների կատարման *առաջնահերթությունը* նվազում է ըստ վերը բերված հաջորդականության: Այսպիսով, ամենաառաջնահերթը *ուևար* գործողություններն են, իսկ հարաբերման գործողություններն իրականացվում են ամենավերջին հերթին:

Նշենք, որ անհրաժեշտության դեպքում *փակագծերի* միջոցով կարելի է գործողությունների կատարման *առաջնահերթությունը փոխել*: Եթե արտահայտության մեջ ներառված գործողությունները կատարման միևնույն առաջնահերթությունն ունեն, ապա իրագործվում են հաջորդաբար՝ ձախից աջ:

Օրինակ՝ $x + y/2$ արտահայտության արժեքը հաշվելիս նախ y -ը կբաժանվի 2-ի վրա, ապա ստացվածը կավելացվի x -ին: Իսկ $(x + y)/2$ -ի դեպքում նախ կիրագործվի $x + y$ գումարի հաշվարկը, ապա արդյունքը կբաժանվի 2-ի վրա: $x * y/2$ արտահայտության արժեքը հաշվելու համար (*Պասկալում* * -ը կիրառվում է որպես բազմապատկման նշան) գործողությունները կիրականացվեն ձախից աջ՝ նախ կբազմապատկվեն x և y փոփոխականների արժեքները, ապա ստացված արտադրյալը կբաժանվի 2-ի:

Այժմ բերենք վերը թվարկված գործողությունների համառոտ բացատրությունները:

Ունար.

+ դրական թիվ,

- բացասական թիվ,

NOT տրամաբանական բացասում, ընդ որում՝ $NOT(TRUE)=FALSE$, իսկ $NOT(FALSE)=TRUE$,

@ օպերանտի հասցե:

Մուտիպլիկատիվ.

***** բազմապատկում (եթե մասնակից օպերանտներից որևէ մեկն իրական է՝ արդյունքն իրական թիվ է),

/ բաժանում (անկախ օպերանտների տիպից՝ արդյունքն իրական թիվ է),

DIV ամբողջ տիպի օպերանտների բաժանում (արդյունքը նույնպես ամբողջ է),

MOD ամբողջ տիպի օպերանտների բաժանման արդյունքի ամբողջ մնացորդի ստացում,

AND տրամաբանական **և** (արդյունքը հավասար է $TRUE$ միայն այն դեպքում, երբ բոլոր մասնակից օպերանտները $TRUE$ արժեք ունեն),

SHL ամբողջ տիպի օպերանտի պարունակության բիթային տեղաշարժ դեպի ձախ (ազատված բիթերը լրացվում են 0-ներով),

SHR ամբողջ տիպի օպերանտի պարունակության բիթային տեղաշարժ դեպի աջ (ազատված բիթերը լրացվում են 0-ներով):

Ադիտիվ.

+ գումարում. արդյունքն իրական է, եթե գումարելիներից մեկն իրական է,

- հանում. արդյունքն իրական է, եթե օպերանտներից մեկն իրական է,

OR տրամաբանական **կամ**. արդյունքը $TRUE$ է, եթե գործողությանը մասնակցող օպերանտներից որևէ մեկն ունի $TRUE$ արժեք,

XOR **բացառող կամ**. արդյունքը $TRUE$ է, եթե օպերանտները տրամաբանական տարբեր արժեքներ ունեն:

Հարաբերման գործողություններ.

= ստուգում է օպերանտների հավասարությունը. արդյունքը $TRUE$ է, եթե օպերանտները հավասար են, հակառակ դեպքում՝ $FALSE$ է,

< եթե անհավասարման ձախ մասի արժեքը փոքր է աջ մասի արժեքից՝ վերադարձվում է $TRUE$, հակառակ դեպքում՝ $FALSE$,

> եթե անհավասարման ձախ մասի արժեքը մեծ է աջ մասի արժեքից՝ վերադարձվում է $TRUE$, հակառակ դեպքում՝ $FALSE$,

<> անհավասարման ստուգում. արդյունքը $TRUE$ է, եթե օպերանտների արժեքներն իրար հավասար չեն,

<= եթե անհավասարման ձախ մասի արժեքը փոքր է կամ հավասար աջ մասի արժեքին՝ վերադարձվում է $TRUE$, հակառակ դեպքում՝ $FALSE$,

>= եթե անհավասարման ձախ մասի արժեքը մեծ է կամ հավասար աջ մասի արժեքին՝ վերադարձվում է $TRUE$, հակառակ դեպքում՝ $FALSE$:

Այն արտահայտությունները, որոնք պարունակում են NOT, AND, OR, XOR կամ հարաբերման գործողություններ՝ պրամաբանական արժեք ունեն: Նման արտահայտությունները կոչվում են պրամաբանական արտահայտություններ:

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ *IN* հարաբերման գործողությունը կիրառվում է երկու օպերանդների նկատմամբ, որպեսզի *IN*-ից չսխալ ընկած մասում եղած արգումենտը պետք է լինի ցանկացած կարգային տիպի, իսկ աջինը՝ նույն տիպի բազմություն կամ իդենտիֆիկատոր:
- ◆ *NOT, AND, OR, XOR* պրամաբանական գործողությունները կիրառելի են նաև ամբողջ տիպի օպերանդների համար. այս դեպքում արդյունքը նույնպես ամբողջ թիվ է (գործողություններն իրականացվում են օպերանդների համապատասխան բիթերի հեղձ՝ ըստ 1.4 աղյուսակի):

Աղյուսակ 1.4

Տրամաբանական (կարգային) գործողություններ ամբողջ տիպի հետ					
I օպերանդ	II օպերանդ	<i>NOT</i>	<i>AND</i>	<i>OR</i>	<i>XOR</i>
1	-	0	-	-	-
0	-	1	-	-	-
0	0	-	0	0	0
0	1	-	0	1	1
1	0	-	0	1	1
1	1	-	1	1	0



1. $2\cos x + 0.1 \sin x^2$ արտահայտությունը հանրահաշվակա՞ն, քե՞տ պրամաբանական արտահայտություն է:
2. Եթե $x=2$, ապա ի՞նչ արժեքներ կընդունեն հետևյալ արտահայտությունները.
 - ա) $(x > 1) \text{ AND } (x < 7)$,
 - դ) $(x < 3) \text{ XOR } (x > 15)$,
 - բ) $(x > 0) \text{ OR } (x < -5)$,
 - ե) $7 \text{ DIV } x$,
 - զ) $\text{NOT } (x > 1)$,
 - զ) $10 \text{ MOD } x$:

3. Մտորեն բերված արտահայտությունները գրեք Պասկալ լեզվով.

- ա) $x^3 + 8 \sin x \cos 3x + \log_3 4x^2$,
- բ) $(y + 1)(x + (x^2 + 1)^2 \sin(x^2 - 3)) - \text{tg}(y)$,
- զ) $\frac{x^2 - 4}{y^2 + 2} + 2^{\sin x}$,
- դ) $\ln(e^x + 1) + \sqrt{x^2 + 4}$,
- ե) $\sin\left(\frac{3x + 4}{y + 2}\right) + \sqrt{(x + 3)^3}$,
- զ) $x^7 + \sin(\cos(x + y))$:

§ 1.7 ՄԵԿՆԱԲԱՆՈՒԹՅՈՒՆՆԵՐ: ՎԵՐԱԳՐՄԱՆ ՕՊԵՐԱՏՈՐ: ՆԵՐՄՈՒԾՈՒՄ

Ծրագիրն առավել *ընթեռնելի* դարձնելու նպատակով ծրագրավորման լեզուներում հատուկ միջոցներ՝ *մեկնաբանություններ* են կիրառվում: Պասկալում մեկնաբանությունը պայմանանշանների ցանկացած հաջորդականություն է, որն առնվում է ձևավոր $\{ \}$ փակագծերի կամ $(*)$ նշանների մեջ: Օրինակ՝

{Սա Պասկալ լեզվի մեկնաբանություն է:} կամ
*(*Ծրագիրը հաշվում է առաջին 100 պարզ թվերի գումարը*)*:

Մեկնաբանություն կարելի է տալ ծրագրի ցանկացած մասում:

Վերագրման օպերատորը փոփոխականին արժեք տալու համար է: Ընդհանուր դեպքում այն կարելի է ներկայացնել հետևյալ կերպ՝

$$A := B;$$

որտեղ A -ն վերագրման արդյունքում արժեք ստացող փոփոխականն է, իսկ B -ն՝ ցանկացած արտահայտություն, որի արժեքը պետք է լինի նույն տիպի, ինչ A փոփոխականինը:

Վերագրման օպերատորի աջ մասում առկա արտահայտության արժեքը պետք է լինի չափս մասի փոփոխականի տիպի (բացառություն է կազմում այն դեպքը, երբ փոփոխականն իրական տիպի է, իսկ արտահայտության արժեքը՝ ամբողջ, բայց ոչ հսկառակը):

Պետք է հիշել, որ վերագրման արդյունքում A փոփոխականի ունեցած նախկին արժեքը ոչնչանում է: Նկատենք նաև, որ վերագրման օպերատորում պարտադիր կերպով մասնակցող $:=$ պայմանանշանների համակցությունը գրառվում է իրար կից՝ առանց դրանց միջև բացատանիշի: Օրինակ՝

$$\begin{array}{ll} x := 2; & \{1\} \\ x := 5*(4+x); & \{2\} \end{array}$$

Եթե բերվածը դիտարկենք որպես իրար հաջորդող հրամաններ, ապա ըստ $\{1\}$ տողի՝ x փոփոխականը ստանում է 2 արժեքը: $\{2\}$ տողի արդյունքում համակարգիչը նախ x -ի ընթացիկ արժեքը (2) տեղադրելով աջ մասի արտահայտության մեջ՝ կհաշվի $4+x$ -ի արժեքը (6), ապա ստացվածը բազմապատկելով 5-ով՝ արդյունքը կվերագրի x -ին: Այսպիսով, արդյունքում x -ը նախկին (2) արժեքի փոխարեն կստանա 30 արժեքը:

Ծրագրի կատարման ընթացքում հաճախ անհրաժեշտ է լինում փոփոխականներին արժեքներ տալ ստեղծաշարից. այս գործընթացն անվանում են *ընթացիկ փոփոխականների ներմուծում*:

Պատկալում տվյալներ ներմուծելու նպատակով կիրառում են **READ** և **READLN** հրամանները, որոնք կարելի է ընդհանուր ձևով նկարագրել հետևյալ կերպ.

READ (մուտքի փոփոխականների ցուցակ);

READLN (մուտքի փոփոխականների ցուցակ);

Այս հրամաններում առկա փակագծերում պետք է ստորակետով անջատելով՝ թվարկել այն փոփոխականները, որոնց արժեքները պետք է տալ ստեղնաշարից: Օրինակ՝

READ(*A,B,C*); {1} կամ՝

READLN(*A,B,C*); {2}

Համակարգիչը հաջորդաբար կատարելով ծրագրում ներառված հրամանները՝ ներմուծման հրամանն իրականացնելիս ընդհատում է ծրագրի հետագա աշխատանքը և սպասում ստեղնաշարից մուտքի ցուցակում ներառված փոփոխականների քանակին համապատասխան արժեքներ ներմուծելուն: Ընդ որում՝ ներմուծված արժեքները հաջորդաբար վերագրվում են մուտքի ցուցակում թվարկված փոփոխականներին: Տվյալները կարելի է ներմուծել միևնույն տողում՝ դրանք բացատանիչերով միմյանցից փոխանջատելով և ներմուծման գործընթացն ավարտելով *ENTER* ստեղնով, կամ տարբեր տողերում՝ ամեն արժեք ներմուծելուց հետո սեղմելով *ENTER* ստեղնը:

READ և *READLN* հրամանների տարբերությունը տեսնելու համար շարունակենք դիտարկել վերը բերված օրինակը: Եթե *A*, *B* և *C* փոփոխականների արժեքները ներմուծվում են *READ* հրամանով, ապա տվյալները կարելի է ներմուծել ցանկացած եղանակով՝

ա) 5 -7 8 (*ENTER*)

բ) 5 -7 (*ENTER*)

8 (*ENTER*)

գ) 5 (*ENTER*)

-7 (*ENTER*)

8 (*ENTER*)

Բերված բոլոր դեպքերում *A* փոփոխականը կստանա 5, *B*-ն՝ -7, իսկ *C*-ն՝ 8 արժեք: Այժմ քննարկենք *READLN*-ի կիրառման դեպքը. եթե ներմուծումը կատարվի ա) եղանակով, ապա արդյունքը համարժեք կլինի *READ*-ի արդյունքին, իսկ բ) և գ) եղանակներն այլ արդյունք կտան: Բանն այն է, որ *READLN* հրամանն *ENTER*-ը «համարում է» ներմուծման գործընթացի ավարտ և այդ պատճառով բ) դեպքում *A* և *B* փոփոխականները արժեքներ կստանան, *C*-ն՝ ոչ, իսկ գ) դեպքում արժեք կստանա միայն *A*-ն:

Եթե, օրինակ, փորձենք *A*, *B*, *C* փոփոխականների արժեքները ներմուծել

READLN(*A,B*); {1}

READ(*C*); {2}

հրամաններով և արժեքները ներմուծենք միևնույն տողի վրա, հետևյալ կերպ՝

5 -7 8 (*ENTER*)

սպա $\{1\}$ հրամանի արդյունքում արժեքներ կստանան միայն A -ն և B -ն, իսկ C -ի համար նախատեսված δ արժեքը կանտեսվի: Շարունակելով աշխատանքը՝ համակարգիչն ըստ $\{2\}$ հրամանի կսպասի ևս մեկ արժեքի ներմուծման, որն էլ կվերագրի C -ին:

READLN հրամանն ունի կիրառման ևս մեկ եղանակ՝ առանց մուտքի ցուցակի (դատարկ).

READLN;

որի դեպքում համակարգիչն ընդհատում է ծրագրի կատարման ընթացքը, սպասում ստեղծաշարի ցանկացած ստեղծ սեղմելուն, որից հետո կրկին շարունակում աշխատանքը:

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ *Միևնույն փիլի սահմանազարիչների միջոցով չի կարելի ներդրված մեկնաբանություններ սրեղծել, սակայն մի փիլի սահմանազարիչների մեջ կարելի է մյուս փիլի սահմանազարիչներով մեկնաբանություն ներդնել՝*

{... (*) ...}* կամ

(... { }... *)*:

- ◆ *READ, READLN հրամաններն իրականում սրանդարդ ենթածրագրեր են՝ առավել լայն գործածման հնարավորություններով:*



1. Պասկալում մեկնաբանություններ սրեղծելու քանի՞ եղանակ կա: Որո՞նք են:
2. Ո՞րն է ծրագրում մեկնաբանություններ սրեղծելու նպատակը:
3. Ո՞ր գրառումներն են ճիշտ վերագրումներ.

ա) $a := 5$;

բ) $7 := C$;

գ) $d := 5 * \sin(x)$;

դ) $\sin(x) := 0.5$;

4. Եթե x -ը իրական փոփոխական է, սպա n ՞ր գրառումներն են ճիշտ.

ա) $x := 3 * \ln(5)$;

բ) $X := -2$;

գ) $X := 'C'$;

դ) $x := true$;

5. Ներմուծման քանի՞ հրաման գիտեք:

6. Ո՞րն է *READ* և *READLN* հրամանների տարբերությունը:

§ 1.8 ԱՐՏԱԾՈՒՄ: ԱՐՏԱԾՄԱՆ ՁԵՎԱԶԱԲ

Ծրագրի աշխատանքի ընթացքում ստացված տվյալները էկրանին արտաձևելու նպատակով կիրառում են *WRITE* և *WRITELN* հրամանները, որոնց ընդհանուր տեսքը հետևյալն է.

WRITE(*ելքի ցուցակ*);
WRITELN(*ելքի ցուցակ*);

որտեղ *ելքի ցուցակը* կարող է հաստատուն արժեքներ, փոփոխականներ, ապաքարգերի (' ') մեջ առնված տեքստային ինֆորմացիա, տրամաբանական և թվաբանական արտահայտություններ պարունակել:

WRITE և *WRITELN* հրամանների աշխատանքի տարբերությունը տեսնելու համար քննարկենք հետևյալ օրինակը.

WRITE (4); *WRITE*(5); *WRITE*(6); {1}
WRITELN(4,5); {2}
WRITE(6); {3}

{1} դեպքում մեկ տողի վրա, իրար կից կարտաձվեն ելքի ցուցակներում ներառված 4, 5 և 6 թվերը. արդյունքում էկրանին կտեսնենք 456 թիվը:

{2} դեպքում մեկ տողի վրա կարտաձվեն միայն 4-ն ու 5-ը, կազմելով 45, իսկ 6-ը կարտաձվի հաջորդ տողում (ըստ {3} հրամանի):

Այսպիսով՝ *WRITELN*-ը, ի տարբերություն *WRITE*-ի, իր ելքի ցուցակում ներառվածն արտաձևելուց հետո տողադարձ է իրականացնում: Արտաձևան հրամանի ելքի ցուցակում ապաքարգերի մեջ ներառված ցանկացած ինֆորմացիա արտաձվում է անփոփոխ. օրինակ՝ *WRITE*('Ես աշակերտ եմ'); հրամանով էկրանին կբերվի *Ես աշակերտ եմ*, իսկ *WRITELN*('a= ', 5); հրամանով՝ *a=5* հաղորդագրությունները:

Արտաձվող տվյալները էկրանին առավել ընթեռնելի ներկայացնելու նպատակով կարելի է տվյալների արտաձևան գործընթացը դեկավարել ծրագրային միջոցներով՝ տալով *արտաձևան չհաշտիք*:

WRITE և *WRITELN* հրամանների ելքի ցուցակում յուրաքանչյուր բաղադրիչին հաջորդող վերջակետ (:) նշանից հետո կարելի է տալ համապատասխան տվյալն արտաձևելու համար տրամադրվող դիրքերի ցանկալի քանակը (դաշտի լայնության չափը)՝ հետևյալ կերպ.

WRITE(*x:k*);

որտեղ *x*-ը արտաձվող փոփոխականի արժեքն է, *k*-ն այն դիրքերի քանակը, որը պետք է տրամադրվի *x*-ի արժեքին: Եթե *x*-ը ամբողջ տիպի փոփոխական է, որի թվանշանների քանակը (բացասական թվի դեպքում ներառյալ նաև նշանը) տրված *k*-ից քիչ է, ապա *x*-ի արժեքն արտաձևելիս ձախ մասից կլրացվի համապատասխան քանակությամբ բացատանիշերով: Օրինակ, եթե *a=72*, և տրվել է *WRITE*('a= ', *a : 5*); հրամանը, ապա կարտաձվի *a= _ _ _ 72* ինֆորմացիան, որտեղ *_*-ով նշվել է բացատանիշը:

Եթե *k*-ն ավելի փոքր է, քան արտաձվող թվի թվանշանների քանակը, ապա *k*-ի մե-

ծությունն անտեսվում է, և թիվն արտաձվում է ամբողջությամբ՝ սկսած ընթացիկ դիրքից:

Եթե արտաձվող տվյալն իրական տիպի է, ապա *արտաձման ձևաչափի* չնշելու դեպքում արժեքն արտաձվում է, այսպես կոչված, *էքսպոնենցիալ տեսքով*, այն նախապես բերելով *նորմալ տեսքի*, որտեղ տասնորդական կետից ձախ միայն մեկ թվանշան է գրվում: Օրինակ, եթե a -ն իրական տիպի է, որի արժեքը 52.15 է, ապա դրա արժեքը կարտաձվի 5.215000000E+01 տեսքով, որն, իհարկե, հարմար չէ ընթերցել:

Նման դեպքերում արտաձվող ինֆորմացիան առավել ընթեռնելի դարձնելու նպատակով հնարավորություն կա արտաձման դաշտի լայնությունից գատ տալու նաև տասնորդական կետից հետո պահանջվող թվանշանների քանակը (թվի ճշտության չափը)՝ հետևյալ կերպ՝ *WRITE(x:k:m)*; : Օրինակ, եթե a -ի 52.15 արժեքն արտաձելու համար կիրառվեր *WRITE('a=',a:10:2)*; հրամանը, ապա արտաձվող մեծությանը կտրվի 10 դիրք, տասնորդական կետից հետո կարտաձվի միայն 2 նիշ. արդյունքում կտեսնենք

$a = \text{ } \llcorner \llcorner \llcorner \llcorner \llcorner 52.15$

ինֆորմացիան:

Եթե *WRITE(x:k:m)* հրամանով արտաձվող x -ի արժեքում տասնորդական կետին հաջորդող մասը m -ից ավելի շատ թվանշաններ է պարունակում, ապա արտաձվող թիվը կտրացվում է՝ ըստ մաթեմատիկայում ընդունված օրենքների:

Օրինակ՝ եթե $x = 101.567$, և այն արտաձվում է *WRITE('x=',x:10:2)*; հրամանով, ապա էկրանին կարտաձվի հետևյալը՝

$x = \text{ } \llcorner \llcorner \llcorner \llcorner 101.57$:

Եթե m -ի արժեքը մեծ է թվի տասնորդական մասի թվանշանների քանակից, ապա արտաձվելիս թվի վերջում համապատասխան քանակությամբ 0-ներ են կցագրվում:

Օրինակ՝ եթե նախորդ դեպքում տրված լիներ *WRITE('x=',x:10:4)*; հրամանը, կտեսնեինք $x = \text{ } \llcorner \llcorner 101.5670$ պատասխանը:

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ *WRITE, WRITELN հրամաններն իրականում սրանդարտ ենթաձրագրեր են՝ առավել լայն գործածման հնարավորություններով:*
- ◆ *Եթե իրական թիվ արտաձելիս WRITE(x:k:m); հրամանում m-ի արժեքը փոքրի 0, այսինքն WRITE(x:k:0) տեսքով, ապա տասնորդական կետը և թվի դրան հաջորդող կոտորակային մասը չեն արտաձվի:*
- ◆ *Տվյալների ներմուծման գործընթացն առավել հասկանալի դարձնելու նպատակով յուրաքանչյուր ներմուծվող փոփոխականի համար կարելի է նախ կիրառել WRITE հրամանը՝ ելքի ցուցակում ապաքարգերի մեջ փոխադրականի վերաբերյալ ինֆորմացիա փոխադրելով: Օրինակ՝*

WRITE('x='); READ(x);

WRITE('y='); READ(y);



1. Ո՞րն է *WRITE* և *WRITELN* հրամանների աշխատանքի տարբերությունը:

2. *x*-ը ամբողջ փոփոխական է, որի արժեքը 215 է: Ի՞նչ կարգաձևի *WRITE('x=', x:2)* հրամանով՝

ա) $x = 21$

բ) $x = 15$

գ) $x = 215$

3. *y*-ը իրական փոփոխական է, որի արժեքը -16.127 է: Ի՞նչ կարգաձևի *WRITE('y=', y:4:2)*; հրամանով՝

ա) $x = -16.$

բ) $x = -16.1$

գ) $x = -16.12$

դ) $x = -16.13$

ե) $y = -16.127$

§ 1.9 ՊԾԱՅԻՆ ԱԼԳՈՐԻԹՄՆԵՐԻ ԾՐԱԳՐԱՎՈՐՈՒՄ

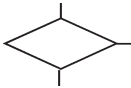
9-րդ դասարանի դասընթացից ծանոթ եք *ալգորիթմ* հասկացությանն ու դրա նկարագրման եղանակներին: Համառոտ վերհիշենք ալգորիթմներից հայտնի նյութը:

Ալգորիթմը քայլերի (գործողությունների) կարգավորված հաջորդականություն է, որը հանգեցնում է սպասված արդյունքին:

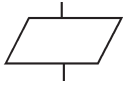
Ալգորիթմ նկարագրելու տարբեր եղանակներից մենք ծանոթ ենք բառաբանաձևային և գրաֆիկական եղանակներին: Քանի որ օգտվելու ենք ալգորիթմի նկարագրման գրաֆիկական եղանակից՝ վերհիշենք դրանում կիրառվող բլոկների նշանակությունները.



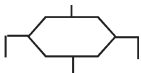
հաշվարկների կատարում և վերագրում,



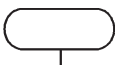
պայմանի ստուգում և հաշվման գործընթացի այլընտրանքային շարունակում,



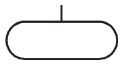
տվյալների ներմուծում, տվյալների արտաձուլում,



ցիկլային գործընթացի կազմակերպում,



ալգորիթմի սկիզբ,



ալգորիթմի սկսարտ,



ալգորիթմի հոսքի ընդհատված մասերի կապի միջոց:

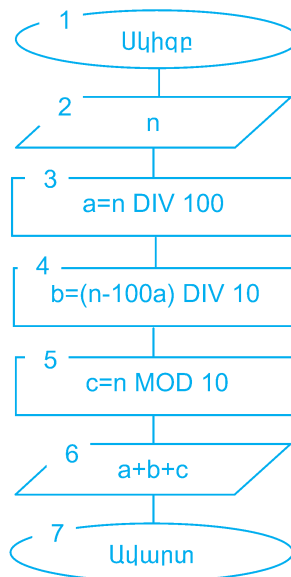
Ալգորիթմները՝ կախված տվյալ պահին լուծվող խնդրից, կարող են լինել **գծային**, **ճյուղավորված** և **ցիկլային**:

Գծային են կոչվում այն ալգորիթմները, որոնցում, պարամետրերի արժեքներից անկախ, գործողությունները կատարվում են միշտ միևնույն հաջորդականությամբ՝ վերից վար, յուրաքանչյուրը՝ միայն մեկ անգամ:

Դիտարկենք հետևյալ խնդիրը.

Տրված է եռանիշ n թիվը: Պահանջվում է հաշվել թիվը կազմող բաղադրիչ թվանշանների գումարը:

Նախ կազմենք խնդրի լուծման բլոկ-սխեման (նկ. 1.3), ապա՝ ծրագիրը:



Նկ. 1.3. Եռանիշ թվի թվանշանների գումարի հաշվման ալգորիթմ

Բերված ալգորիթմում կիրառվել են ամբողջ թվերի համար սահմանված DIV և MOD ստանդարտ ֆունկցիաները, որտեղ $a DIV b$ -ն վերադարձնում է a -ն b -ի բաժանելիս ստացվող ամբողջ արժեքը (օրինակ՝ $7 DIV 3=2$), իսկ MOD -ը՝ այդ բաժանման ամբողջ մնացորդը (օրինակ՝ $7 MOD 3=1$):

Եթե, օրինակ՝ $n=672$, ապա 3-րդ բլոկով կատանանք՝ $a = 672 DIV 100 = 6$ *արժեքը*, 4-րդ բլոկով՝ $b = (672 - 100 \cdot 6) DIV 10 = 7$ *արժեքը*, իսկ 5-րդով՝ $c = 672 MOD 10 = 2$ *արժեքը*:

Այսպիսով, a , b , c փոփոխականների մեջ ստացվել են եռանիշ թվի բաղադրիչները, մնում է 6-րդ բլոկով արտածել պահանջվող գումարը:

Կազմենք ծրագիրը.

PROGRAM Eranish;

VAR n:Word; {n-ը եռանիշ թիվն է:}

a,b,c: BYTE; {a-ն հարյուրավորն է, b-ն՝ տասնավորը, c-ն՝ միավորը}

BEGIN

WRITE('n= '); READ(n); { Եռանիշ թվի՝ n-ի ներմուծում }

a:=n DIV 100; {Հարյուրավորի ստացում}

*b:=(n-100*a) DIV 10; {Տասնավորի ստացում}*

c:=n MOD 10; {Միավորի ստացում}

WRITELN ('n-ի թվանշանների գումարը =',a+b+c:4) {արդյունքի

արտածում}

END.

n-ը հայտարարված է *WORD* տիպի, քանի որ այն պետք է պարունակի դրական ամբողջ թիվ, բայց որը չի կարող բնութագրվել որպես *BYTE*, քանի որ *BYTE*-ում տեղավորվող ամենամեծ թիվը 255-ն է (իսկ եռանիշի վերին եզրը 999 է): *a, b, c* փոփոխականների արժեքները նույնպես դրական ամբողջ թվեր են, որոնց արժեքները չեն կարող 9-ից մեծ լինել. նման արժեքների համար առավել հարմար է *BYTE* տիպը:

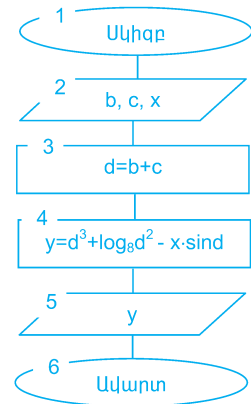
Նկատել, որ ծրագրում բոլոր հրամաններն ավարտվել են ;-ներով, բացի *END*-ին նախորդող հրամանից. պատճառն այն է, որ *END*-ը ոչ միայն սահմանափակում է *BEGIN*-ով սկսված բլոկը, այլև ավարտում իրեն նախորդող օպերատորը: Այսպիսով, կետ-ստորակետ դնելու դեպքում վերջին օպերատորը երկու ավարտ կունենար՝ կետ-ստորակետն ու *END*-ը: Պասկալի կոմպիլյատորը չնայած որևէ սխալ չէր գտնի, սակայն «կմտածեր», թե *END*-ի և վերջին օպերատորի միջև ևս օպերատոր կա. նման հրաման չպարունակող օպերատորն անվանում են **դադարկ օպերատոր**: Դատարկ օպերատոր կա նաև երկու իրար հաջորդող կետ-ստորակետների միջև: Օրինակ՝ *x:=8; ; y:=-7; ;*

Ինչպես երևում է վերը բերված ծրագրից՝ գծային ալգորիթմների օգնությամբ լուծվող հաշվարկային խնդիրները կարող են պարունակել միայն ներմուծման, արտածման հրամաններ և հաշվարկներ կատարելու համար՝ վերագրման օպերատորներ:

Դիտարկենք գծային ալգորիթմով լուծվող ևս մի խնդիր.

x, b, c պարամետրերի ցանկացած իրական արժեքների համար հաշվել և արտածել y-ի արժեքը, եթե $y = (b + c)^3 + \log_8(b + c)^2 - x \sin(b + c)$:

Նախ կառուցենք խնդրի լուծման բլոկ-սխեման (նկ. 1.4).



Նկ. 1.4. $y = (b + c)^3 + \log_8(b + c)^2 - x \sin(b + c)$
արգասահայրության արժեքի հաշվման ալգորիթմ

3-րդ բլոկում մտցված լրացուցիչ d փոփոխականի մեջ պահվել է $b + c$ արտահայտության արժեքը, որպեսզի 4-րդ բլոկում ներառված արտահայտության արժեքը հաշվարկելիս $b + c$ -ի արժեքը մի քանի անգամ չհաշվենք: Գրենք ծրագիրը.

```
PROGRAM Hashvark;
VAR d,b,c,x,y:REAL;           {1}
BEGIN
    WRITE('b= '); READ(b);
    WRITE('c= '); READ(c);
    WRITE('x= '); READ(x);
    d:=b+c;
    y:=EXP(3*LN(d))+LN(SQR(d))/LN(8)-x*SIN(d) ;      {2}
    WRITELN('y= ',y :8 :3)
END.
```

{1} տողում հայտարարվել են խնդրի լուծման գործընթացում կիրառվող բոլոր փոփոխականները, ընդ որում՝ b , c , x պարամետրերն իրական տիպի են՝ ըստ խնդրի պահանջի, d -ն պետք է պարունակի $b+c$ -ի արժեքը, որը նույնպես իրական կլինի, իսկ y -ն իրական է, քանի որ պետք է ստացվի իրական արժեք ներկայացնող արտահայտությունից:

{2} տողում կատարված հաշվարկի մեջ d^3 -ը հաշվվել է $EXP(3*LN(d))$ բանաձևով՝ ըստ $a^b = e^{b \cdot \ln a}$ մաթեմատիկական առնչության, իսկ $\log_8 d^2$ -ն հաշվվել է լոգարիթմի մի հիմքից այլ հիմքին անցնելու կանոնով (քանի որ Պասկալում հայտնի է միայն բնական հիմքով լոգարիթմը) և 8 հիմքով լոգարիթմից անցում է կատարվել բնական հիմքի՝

$$\log_8 d^2 = \frac{\ln d^2}{\ln 8} = \ln(\text{sqr}(d)) / \ln(8) :$$

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ **Ծրագիր կազմելիս պահանջվող փվյալների միշտ հայտարարությունը կարևորվում է նրանով, որ համակարգի օպերատիվ հիշողությունն անհրաժեշտ է ռացիոնալ օգտագործել:**



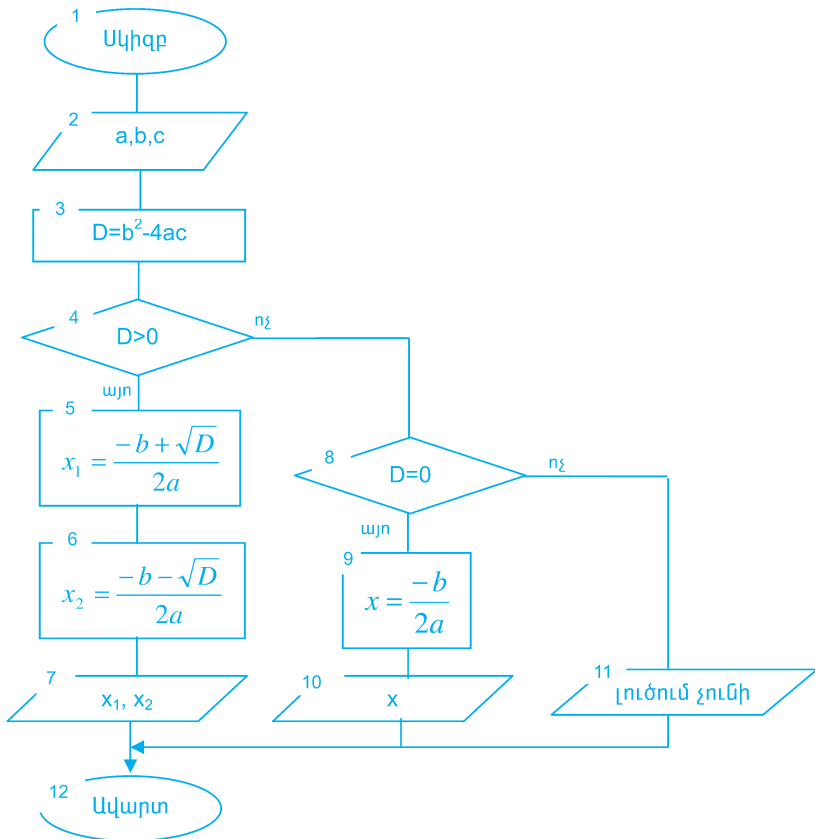
1. **Գծային ալգորիթմներում n^2 հրամաններն ու օպերատորներն են կիրառվում:**
2. **Ո՞րն է դպարակ օպերատորը, ինչպե՞ս է այն կազմվում:**
3. **Կազմել հավելված 3-ի այս թեմային առնչվող խնդիրների լուծման բլոկ-սխեմաներն ու ծրագրերը:**

§ 1.10 ՃՅՈՒՂԱՎՈՐՄԱՆ ԳՈՐԾԸՆԹԱՅՆ ԱԼԳՈՐԻԹՄՆԵՐՈՒՄ: ՃՅՈՒՂԱՎՈՐՄԱՆ (ՊԱՅՄԱՆԻ) ՕՊԵՐԱՏՈՐ

Գործնականում խնդիրների լուծման ալգորիթմները *ճյուղավորումներ* են պարունակում. դա պայմանավորվում է լուծման մեջ առկա պայմաններով, որոնցից կախված խնդրի հետագա լուծումը կարող է շարունակվել տարբեր ուղիներով:

Հիշենք, որ այն ալգորիթմը, որտեղ ստուգվող պայմանից կախված խնդրի լուծման գործընթացը շարունակվում է տարբեր ուղիներով, անվանում են *ճյուղավորված*, իսկ սվյալ ուղիները՝ *ճյուղեր*:

Որպես օրինակ *դիտարկենք* $ax^2 + bx + c = 0$ *քառակուսի հավասարման* ($a \neq 0$) *իրական արմատները գտնելու ալգորիթմը*:



Նկ. 1.5. Քառակուսի հավասարման իրական արմատները փնտրելու ալգորիթմը

Բլոկ-սխեմայից երևում է, որ 4-րդ բլոկում ներառված պայմանի ճշմարիտ (TRUE) կամ կեղծ (FALSE) լինելուց կախված՝ խնդրի լուծումը շարունակվում է տարբեր ուղի- ուղություններով: Ընդ որում՝ պայմանի կեղծ լինելու դեպքում, ըստ 8-րդ բլոկում ներառ-

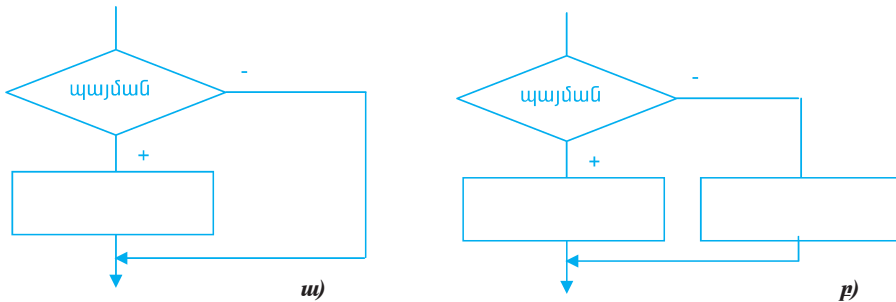
ված պայմանի ընդունած արժեքի՝ ալգորիթմի մեջ մեկ այլ ճյուղավորում է առաջանում:

Ճյուղավորումներ պարունակող ալգորիթմները ծրագրավորելու նպատակով Պասկալում **պայմանի օպերատոր** է նախատեսված, որն ունի հետևյալ ընդհանուր տեսքը.

IF a THEN b ELSE c;

որտեղ *a*-ն տրամաբանական արտահայտություն է, *b*-ն և *c*-ն՝ ցանկացած օպերատորներ կամ *BEGIN* ու *END* բառերի միջև առնված օպերատորների համախմբություն՝ բլոկ:

Եթե պայմանի օպերատորը ծրագրավորում է նկ.1.6 ա-ում բերված գործընթացը, ապա կիրառվում է **պայմանի օպերատորի համառոտ** տեսքը՝ *IF a THEN b;* , իսկ նկ. 1.6 բ-ում բերվածի դեպքում *IF a THEN b ELSE c;* տեսքի պայմանի օպերատորը, որն անվանում են **ընդարձակ**:



Նկ. 1.6. Ալգորիթմների ճյուղավորումը

Այժմ կազմենք քառակուսի հավասարման արմատների որոշման ալգորիթմին (նկ. 1.5) համապատասխանող ծրագիրը.

```

PROGRAM Qar_Havasarum;
VAR a,b,c,z,d:REAL;
    x1,x2,x:REAL;
BEGIN WRITE('a= '); READ(a);
      WRITE('b= '); READ(b);
      WRITE('c= '); READ(c);
      d:=SQR(b)-4 * a * c;           {1}
      z:=2 * a;                     {2}
      IF d>0 THEN
BEGIN
      x1:=(-b-SQRT(d)) / z;
      x2:=(-b+SQRT(d)) / z;
      WRITELN('x1=',x1:10:2,'  ' 'x2=',x2:10:2)   {3}
END

```

```

ELSE IF d=0 THEN
  BEGIN
    x := - b / z;
    WRITELN('հավասարումը մեկ արմատ ունի՝ x= ',x:10:2)
  END
ELSE WRITELN('հավասարումն իրական թվերի
  բազմության մեջ լուծում չունի')
END.

```

Ծրագրի {1} տողում հաշվարկվել ու d -ի մեջ պահպանվել է քառակուսի հավասարման որոշիչը, իսկ {2} տողում z -ին վերագրվել է $2*a$ -ի արժեքը՝ չնայած բլոկ-սխեմայում դրա անհրաժեշտությունը չէր զգացվում: Բանն այն է, որ $2*a$ -ի արժեքը ծրագրում անհրաժեշտ է եղել կիրառել մի քանի անգամ և մեկ անգամ հաշվելով ու պահպանելով՝ ծրագրի արագագործությունը դրանից միայն կշահի:

Դիտելով նկ. 1.5-ում բերված բլոկ-սխեման, նկատենք, որ 4-րդ և 8-րդ բլոկներում ներառված պայմաններից յուրաքանչյուրի ճշմարիտ լինելու դեպքում պետք է իրագործվեն մեկից ավելի գործողություններ: Պայմանի ճշմարիտ կամ կեղծ լինելու դեպքում, երբ անհրաժեշտություն է առաջանում մեկից ավելի օպերատորներ իրականացնել, այդ օպերատորներից կազմվում է բլոկ (համապատասխան օպերատորներն առնվում են *BEGIN* և *END* բառերի մեջ):

{3} մեկնաբանությամբ տողում կիրառված *WRITELN* հրամանով $x1$ և $x2$ արմատների արժեքներն արտածվում են միևնույն տողի վրա: Որպեսզի դրանք իրարից փոխանջատվեն, արտածվող պարամետրերի միջև մտցվել է ‘**┃**’ : 3 պարամետրը, ըստ որի $x1$ -ի և $x2$ -ի արժեքներն իրարից կտարանջատվեն 3 իրար կից բացատանիշերով:

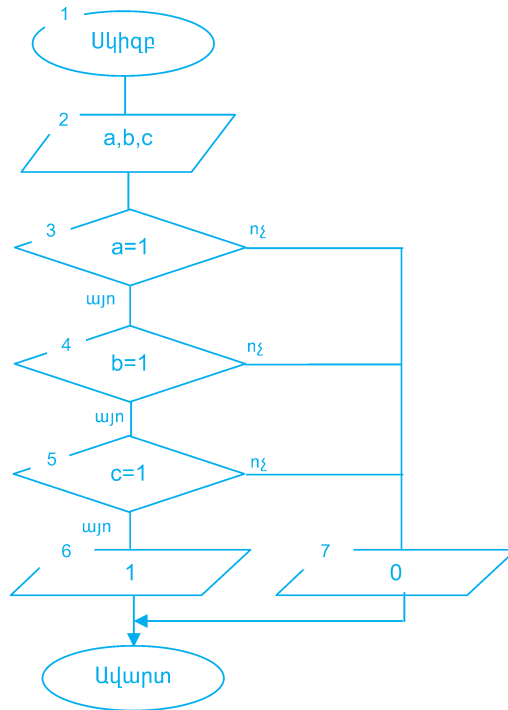
Քննարկենք հետևյալ խնդիրը. *եթե տրված a , b և c ամբողջ տիպի փոփոխականների արժեքները հավասար են 1-ի՝ արտածել 1, հակառակ դեպքում (եթե դրանցից թեկուզ մեկը հավասար չէ 1-ի) արտածել 0 թիվը:*

Փորձենք խնդրի պահանջը ձևակերպել մեկ այլ եղանակով՝ եթե միաժամանակ ճիշտ են $a=1$ և $b=1$ և $c=1$ պայմանները, այլ խոսքով՝ եթե նշված պայմանները *համակարգելի են*՝ արտածել 1, հակառակ դեպքում՝ 0: *Համակարգելի պայմանները* բլոկ-սխեմայում իրար կցվում են պայմանների ճշմարիտ (*այո*) ճյուղերի ուղղություններով:

Այսպիսով, խնդրի պահանջն արտահայտվում է մի քանի պայմանների համադրմամբ՝ ունենք բաղադրյալ պայման, որտեղ *պարզ պայմաններն* իրար են կցվելու *AND* առանցքային բառի միջոցով:

AND-ի միջոցով կազմավորված բաղադրյալ պայմանը ճշմարիտ է, եթե ճշմարիտ են բոլոր բաղադրիչ պայմանները և դրանցից թեկուզ մեկի կեղծ լինելու դեպքում կեղծ է ամբողջ բաղադրյալ պայմանը:

Կառուցենք խնդրի լուծման բլոկ-սխեման (նկ. 1.7):



Նկ. 1.7. ճյուղավորված ալգորիթմի օրինակ

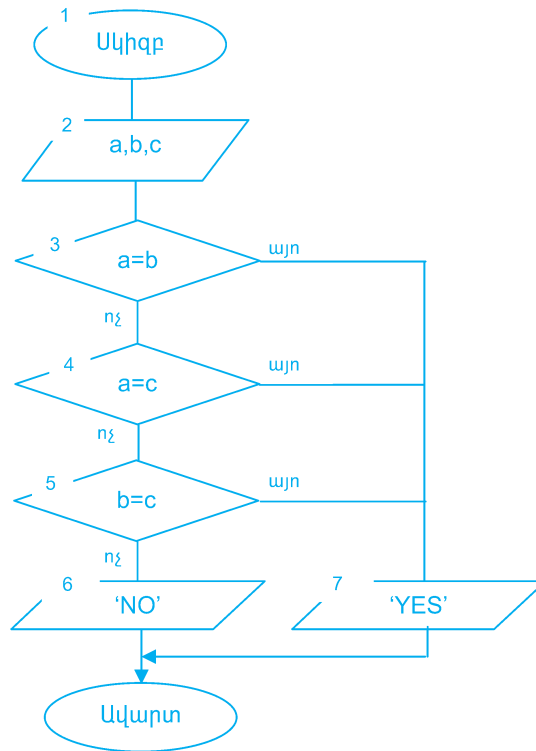
Կազմենք ծրագիրը.

```

PROGRAM Baxadryal;
VAR a,b,c:BYTE;
BEGIN  WRITE('a= '); READ(a);
        WRITE('b= '); READ(b);
        WRITE('c= '); READ(c);
        IF (a=1)AND(b=1)AND(c=1) THEN WRITELN(1)
        ELSE WRITELN(0)
END.
  
```

Այժմ լուծենք հետևյալ խնդիրը. արտածել YES բառը, եթե a, b, c իրական քվե-րի մեջ գոյություն ունեն իրար հավասար քվանդաններ, հակառակ դեպքում՝ NO հաղորդագրությունը:

Կազմենք խնդրի բլոկ-սխեման:



Նկ. 1.8. Ճյուղավորված ալգորիթմի օրինակ

Այս խնդրի պահանջը ևս փորձենք այլ կերպ ձևակերպել. եթե ճշմարիտ է $a=b$ կամ $a=c$, կամ $b=c$ պայմաններից գոնե մեկը, այլ խոսքով, եթե ունենք **համախմբելի պայմաններ**, ապա արտածել *YES*, հակառակ դեպքում՝ *NO*: Համախմբելի պայմանները բլոկ-սխեմայում իրար կցվում են պայմանների *կեղծ* (ոչ) ճյուղերի ուղղություններով:

Այսպիսով, վերը բերվածը բաղադրյալ պայման է, որտեղ պարզ պայմաններն իրար կցվելու են **OR** առանցքային բառի միջոցով:

OR-ի միջոցով կազմավորված բաղադրյալ պայմանը ճշմարիտ է, եթե ճշմարիտ է այն կազմող բաղադրիչ պարզ պայմաններից գոնե մեկը:

Կազմենք նկ. 1.8-ում բերված ալգորիթմին համապատասխանող ծրագիրը.

```

PROGRAM Yes_No;
VAR a,b,c:REAL;
BEGIN  WRITE('a = '); READ(a);
        WRITE('b = '); READ(b);
  
```

```
WRITE('c = '); READ(c);
IF (a = b)OR(a = c)OR(b = c) THEN WRITELN('YES')
ELSE WRITELN('NO')
```

END.

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ Բաղադրյալ պայմանը ծրագրավորելիս այն կազմող յուրաքանչյուր պարզ պայման պետք է առնել փակագծերի () մեջ:
- ◆ Եթե պայմանի օպերատորները ներդրված են, ապա հերթական ELSE-ը համապատասխանում է դրան նախորդող մոտակա IF-ին:



1. Ո՞ր ալգորիթմն է համարվում ճյուղավորված:
2. Պատկարում ճյուղավորված ալգորիթմների ծրագրավորման նպատակով ի՞նչ օպերատոր են կիրառում:
3. Քանի՞ հնարավոր տեսք ունի պայմանի օպերատորը, ինչպե՞ս են դրանք կոչվում:
4. Բաղադրյալ պայմանը կազմող համախմբելի պարզ պայմանների բլոկները ո՞ր ճյուղերով են կցվում միմյանց և ինչպե՞ս են ծրագրավորվում:
5. Տրված են երեք դրական բվեր: Արտածել YES, եթե այդպիսի երկարություններ ունեցող հարվածներով հնարավոր է եռանկյունի կառուցել, հակառակ դեպքում՝ NO հաղորդագրությունը:
6. Տրված են երեք բվեր: Արտածել YES, եթե տրված բվերից գոնե մեկը զույգ է, այլապես՝ NO հաղորդագրությունը:
7. Տրված են երեք բվեր: Արտածել YES, եթե այդ բվերի մեջ կան իրար հակադիր բվեր, հակառակ դեպքում՝ NO հաղորդագրությունը:
8. Տրված է քառանիշ թիվ: Հաշվել և արտածել քառանիշ բվի հարաբերության արժեքը հազարավորների և միավորների բվանշանների գումարին, եթե քառանիշ թիվը փոքր է 5000-ից, հակառակ դեպքում արտածել քառանիշ թիվը:
9. Տրված է եռանիշ թիվ: Հաշվել և արտածել եռանիշ բվի բվանշաններից փոքրագույնի արժեքը:
10. Կազմել հավելված 3-ի այս թեմային առնչվող խնդիրների լուծման բլոկ-սխեմաներն ու ծրագրերը:

§ 1.11 ԸՆՏՐՈՒԹՅԱՆ ՕՊԵՐԱՏՈՐ

Ընտրության օպերատորն իր գործողությամբ նման է պայմանի օպերատորին: Այս օպերատորի ընդհանուր տեսքը հետևյալն է.

```

CASE a OF
  c1: b1;
  c2:b2;
  .
  .
  .
  cn: bn
ELSE BNN;
END;

```

որտեղ՝

a-ն կարգային փիպի արժեք ունեցող արտահայտություն է,
c1...cn -ը կարգային փիպի հասարակություններ են,
b1... bn, bnn -ը ցանկացած օպերատորներ են կամ բլոկ:

Ընտրության օպերատորն աշխատում է հետևյալ կերպ. համակարգիչը հերթով, վերից վար ստուգում է, թե *a* արտահայտության արժեքը *c1, c2, ..., cn* արժեքներից որի հետ է համընկնում. եթե այդպիսի արժեք կա թվարկվածների մեջ, ապա իրագործվում է այդ արժեքին երկու կետից (:) հետո հաջորդող օպերատորը կամ բլոկը, հակառակ դեպքում՝ *ELSE*-ին հաջորդողը:

Այս գործընթացում որոշիչ դեր ունեցող արտահայտությունը՝ *a*-ն, անվանում են *ընտրիչ*:

Ընտրության օպերատորում *ELSE* բաղադրիչը կարող է բացակայել. այս դեպքում, եթե թվարկված հաստատուններից ոչ մեկի արժեքը չի համընկնում ընտրիչ արտահայտության արժեքի հետ, ապա ընտրության օպերատորն ուղղակի դիտարկվում է որպես դատարկ օպերատոր և ծրագրի կատարման ընթացքը հանձնվում է *CASE*-ն ավարտող *END*-ին հաջորդող օպերատորին:

Եթե *CASE*-ի ընտրիչի մի քանի արժեքների դեպքում պետք է իրականացվի միևնույն օպերատորը կամ բլոկը, ապա այդ արժեքները կարելի է թվարկել՝ իրարից անջատելով ստորակետերով, վերջում դնել : նշանն ու տալ համապատասխան օպերատորը կամ բլոկը, օրինակ՝ հետևյալ կերպ.

```

CASE a OF
  c1: b1;
  c2, c3, c4:b2;
  .
  .
  .
  cn: bn
ELSE BNN;
END;

```


Ընտրության օպերատորի աշխատանքին ծանոթանալու նպատակով դիտարկենք հետևյալ խնդիրը. *ներմուծված սիմվոլային s պայմանանշանի +, -, *, / արժեքների դեպքում արտածել համապատասխանաբար գումարում, հանում, բազմադասկում և բաժանում բառերը, ներմուծված այլ պայմանանշանների դեպքում արտածել 'սա թվաբանական գործողություն չէ' տեքստը:*

```
PROGRAM Case_Select;
VAR s:CHAR;
BEGIN WRITE('s= '); READLN(s);
      CASE s OF
        '-': WRITE('հանում');
        '+': WRITE('գումարում');
        '*': WRITE('բազմապատկում');
        '/': WRITE('բաժանում')
          ELSE WRITELN('սա թվաբանական գործողություն չէ')
        END
      END.
```

Ծրագիրը նախ ստեղծարարից ներմուծում է սիմվոլային տիպի s փոփոխականի արժեքը, ապա ընտրության CASE օպերատորը սկսում է վերից վար հերթով ստուգել, թե '+', '-', '*', '/' պայմանանշաններից որի հետ է համընկնում ներմուծվածը: Եթե մեկնումեկի հետ համընկնում է, ապա իրագործվում է համապատասխան տողում գրված արտածման հրամանն ու ընտրության օպերատորն ավարտում է աշխատանքը, իսկ եթե չի համընկնում նշված արժեքներից ոչ մեկի հետ՝ իրագործվում է CASE-ում ներառված ELSE-ին հաջորդող արտածման հրամանը:

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ CASE-ում կիրառված ELSE-ից առաջ ավարտված օպերատորը կարելի է վերջացնել ;-ով:
- ◆ Եթե CASE-ի ընկրիչի որևէ արժեքի դեպքում անհրաժեշտ է մեկից ավելի օպերատորներ իրականացնել, ապա դրանք պետք է առնել BEGIN և END առանցքային բառերի մեջ:
- ◆ Սիմվոլային փոփոխականի արժեքը պետք է ծրագրում անպայման ներմուծել READLN հրամանի միջոցով:



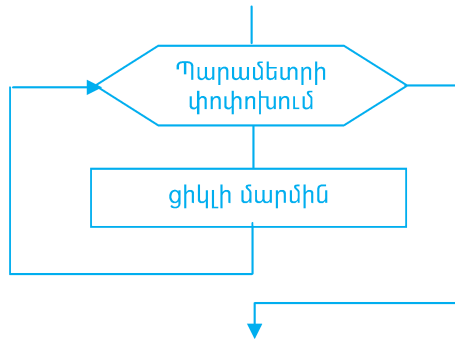
1. Ընկրության օպերատորն իր իմաստով չեզ հայտնի n^ր օպերատորին է նման:
2. Ծրագրի հեղեյալ հասցվածը փոխարինեք գործողության առումով դրան համազոր պայմանի օպերատորով.

```
.....
CASE k OF
  1: WRITE(1);
  2: WRITE(2);
  ELSE WRITE(0)
END;
```

3. Ըստ սրեղնաշարից ներմուծված x և y իրական քվերի և c պայմանանշանի արժեքների՝ գրել հեղուկայ խնդրի լուծման ծրագիրը. եթե c պայմանանշանը
 ‘+’ է՝ հաշվել և արտածել $x + y$ -ի արժեքը,
 ‘-’ է՝ հաշվել և արտածել $x - y$ -ի արժեքը,
 ‘*’ է՝ հաշվել և արտածել $x * y$ -ի արժեքը,
 / է, հաշվել և արտածել x / y -ի արժեքը,
 իսկ այլ արժեքի դեպքում՝ արտածել ‘Միայլ գործողություն է ներմուծվել’
 տեքստը:

§ 1.12 ԿՐԿՆՈՒԹՅԱՆ (ՑԻԿԼԻ) ՕՊԵՐԱՏՈՐՆԵՐ

9-րդ դասարանի դասընթացից արդեն ծանոթ եք կրկնողական (շրջափուլային) կամ, այսպես կոչված, ցիկլային բնույթ կրող գործառույթներին: Վերհիշենք, որ ալգորիթմներում գործողությունը կամ գործողությունների խումբը որոշակի անգամ կրկնելու նպատակով կիրառում են պարամետրով ցիկլային կառուցվածքներ, որոնք բլոկ-սխեմաներում ներկայացվում են մոդիֆիկացիայի բլոկի միջոցով տրվող ընդհանրական սխեմայով (նկ. 1.9):



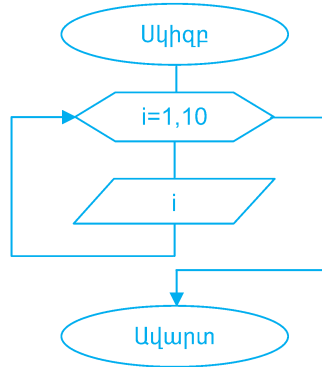
Նկ. 1.9. Պարամետրով ցիկլային կառուցվածքի ընդհանրական սխեմա

Կրկնողական գործընթացներին ծանոթանալու նպատակով դիտարկենք հետևյալ խնդիրը՝ *արտածել 1-ից 10 միջակայքի ամբողջ թվերը*:

Խնդրի լուծման բլոկ-սխեման բերված է նկ. 1.10-ում:

Այստեղ ցիկլում ներառված միակ գործողության (i փոփոխականի արտածում) կրկնությունների քանակը հայտնի է՝ 10 անգամ: Այսպիսի գործառույթները ծրագրավորելու համար *պարամետրով ցիկլի օպերատոր* են կիրառում, որը *Պասկալում* ունի հետևյալ ընդհանուր տեսքը.

FOR ցիկլի պարամետր := ցիկլի պարամետրի առաջին արժեք TO ցիկլի պարամետրի վերջնական արժեք DO ցիկլի մարմին;



Նկ. 1.10. 1-ից 10 միջակայքի ամբողջ թվերի արտածման բլոկ-սխեմա

Այս օպերատորն անվանում են **ցիկլի աճող պարամետրով** օպերատոր: Այստեղ *FOR*, *TO* և *DO* բառերն առանցքային բառեր են:

Գրենք նկ. 1.10-ում բերված բլոկ-սխեմային համապատասխանող ծրագիրը.

```

PROGRAM Par_Cikli;
VAR i:BYTE;
BEGIN
    FOR i:=1 TO 10 DO           {1}
        WRITELN(i)           {2}
    END.
  
```

{1} տողում գրվածի համաձայն՝ ծրագրի սկզբում ցիկլի *i* պարամետրը ստանում է *I* նախնական արժեք, որը համեմատվում է պարամետրի վերջնական արժեք հանդիսացող 10-ի հետ: Եթե պարզվեր, որ պարամետրի նախնական *I* արժեքն ավելի մեծ է, քան վերջնական 10 արժեքը, ապա ցիկլը (ոչ մի անգամ չիրագործելով ցիկլի մարմնում ներառված հրամանը) կավարտեր աշխատանքը: Քանի որ այդպես չէ՝ իրականացվում է {2} տողում ներառված հրամանը՝ արտածելով *i*-ի ընթացիկ արժեքը՝ *I* թիվը: Այնուհետև վերադարձ է կատարվում ցիկլի սկիզբ՝ ավտոմատ կերպով (ըստ պարամետրով ցիկլի գործողության սկզբունքի) նախապես *I*-ով մեծացնելով ցիկլի *i* պարամետրի ընթացիկ արժեքը, և ամեն ինչ (պարամետրի ընթացիկ արժեքի համեմատումը վերջնականի հետ և այլն) կրկնվում է նորից: Յիկլն ավարտում է աշխատանքը, երբ *i* պարամետրի արժեքը դառնում է հնարավոր վերջնականից՝ 10-ից մեծ:

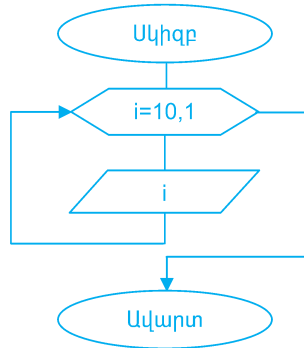
Եթե անհրաժեշտ լիներ 10-ից մինչև 1-ն ընկած թվերն արտածել 10, 9, 8, ..., 1 հաջորդականությամբ, ապա այս գործընթացը (երբ ցիկլի պարամետրը նվազելով է հասնում վերջնական արժեքին) ծրագրավորվում է ցիկլի **նվազող պարամետրով օպերատորի** միջոցով, որի տեսքն այսպիսին է՝

```

FOR ցիկլի պարամետր := ցիկլի պարամետրի առաջին արժեք DOWNTO
    ցիկլի պարամետրի վերջնական արժեք DO ցիկլի մարմին;
  
```

Այստեղ կիրառված *DOWNTO*-ն նույնպես առանցքային բառ է:

Բերենք 10-ից 1 թվերն արտածելու խնդրի բլոկ-սխեման ու ծրագիրը:



Նկ. 1.11. 10-ից 1 միջակայքի ամբողջ թվերի արտածման բլոկ-սխեմա

```

PROGRAM Par_Cikl2;
VAR i:BYTE;
BEGIN
    FOR i:= 10 DOWNT0 1 DO           {1}
        WRITELN(i)
    END.
  
```

Ծրագրի {1} տողում նախ i պարամետրը ստանում է 10 արժեքը. ցիկլի օպերատորը DOWNT0-ով գրելու դեպքում ցիկլը աշխատանքը կավարտեր այս փուլում, եթե պարզվեր, որ պարամետրի ընթացիկ արժեքը փոքր է վերջնականից: Քանի որ 10-ը փոքր չէ 1-ից՝ իրագործվում է ցիկլի մարմինը՝ արտածելով i -ի արժեքը (10), որից հետո վերադարձ է կատարվում նորից ցիկլի սկիզբ՝ այս անգամ արդեն պարամետրի ընթացիկ արժեքը 1-ով ավտոմատ պակասեցնելով: Ամբողջ գործընթացը նորից կրկնվում է սկզբից՝ պարամետրի ընթացիկ 9 արժեքը համեմատվում է վերջնականի՝ 1-ի հետ և այլն:

Այժմ կրկնողական բնույթի մեկ այլ գործընթաց ուսումնասիրենք: Ենթադրենք՝ անհրաժեշտ է արտածել ստեղնաշարից ներմուծված պայմանանշանի կողմ այնքան ժամանակ, քանի դեռ չի ներմուծվել ‘ / ‘ պայմանանշանը:

Կազմենք այս գործընթացն ապահովող ծրագրի բլոկ-սխեման (նկ. 1.12):

Այս բլոկ-սխեմայով նույնպես կրկնողական գործընթաց է իրականացվել, սակայն, ի տարբերություն քննարկված նախորդ դեպքերի, այստեղ գործողությունների կրկնության քանակն անհայտ է: Նման գործընթացներ ծրագրավորելու համար *Պասկալում* նախատեսված են ցիկլի երկու՝ *նախապայմանով* ու *վերջնապայմանով* օպերատորներ:

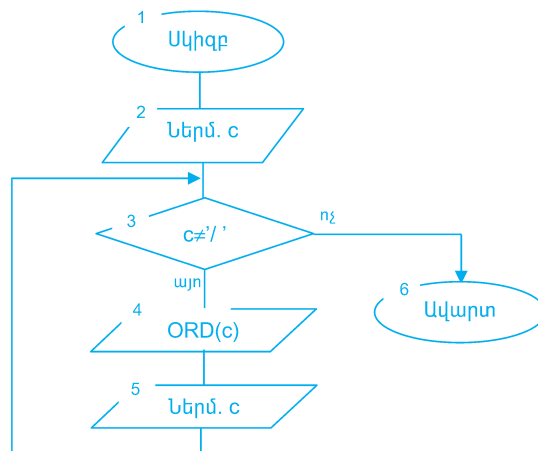
Դիտարկենք *ցիկլի նախապայմանով օպերատորը*, որի ընդհանուր տեսքը հետևյալն է.

WHILE a DO ցիկլի մարմին;

Այստեղ *WHILE*-ը և *DO*-ն առանցքային բառեր են, *a*-ն տրամաբանական արտահայտություն է, *ցիկլի մարմինը*՝ օպերատոր կամ բլոկ:

Նախապայմանով ցիկլի օպերատորի մարմինը կրկնվելով իրագործվում է այնքան, քանի դեռ ցիկլի կրկնության պայմանի՝ *a* տրամաբանական արտահայտության արժեքը ճշմարիտ (*TRUE*) է, և ավարտվում է այն կեղծ (*FALSE*) դառնալու դեպքում:

Այսպիսով, նախապայմանով ցիկլի մարմինը կարող է ոչ մի անգամ չիրագործվել, եթե *a*-ն ի սկզբանե ունենա *FALSE* արժեք, իսկ մյուս դեպքում էլ անվերջ կրկնվել, եթե *a*-ն երբեք *FALSE* արժեք չստանա:



Նկ. 1.12. Նախապայմանով ցիկլային ալգորիթմի օրինակ

Գրենք նկ. 1.12-ում բերված բլոկ-սխեմային համապատասխանող ծրագիրը.

```

PROGRAM Max_Cikl;
VAR c:CHAR;
BEGIN WRITE('Որևէ տրեղն սեղմեք');
      READLN(c);
      WHILE c <> '/' DO
      BEGIN WRITELN('Սեղմված պայմանանշանի կողմ
        հավասար է', '  ':3, ORD(c));
        READLN(c)
      END
    END.
  
```

{1}

Այսպիսով, {1} տողում ներառված նախապայմանով ցիկլի օպերատորը նախ որոշում է *C<> '/'* տրամաբանական արտահայտության արժեքը, և եթե այն *TRUE* է (ներմուծվածը *'/'* պայմանանշանը չէ), ապա իրականացվում են *BEGIN* և *END*-ի մեջ ներառված օպերատորները. այստեղ ցիկլի մարմինը բլոկ է կազմում, քանի որ կրկնվող մեկ օպերատոր չէ, այլ մեկից ավելի:

Այժմ ծանոթանանք *վերջնապայմանով* կամ, այսպես կոչված, *հեղապայմանով* *ցիկլի օպերատորին*: Սրա ընդհանուր տեսքը հետևյալն է՝

REPEAT

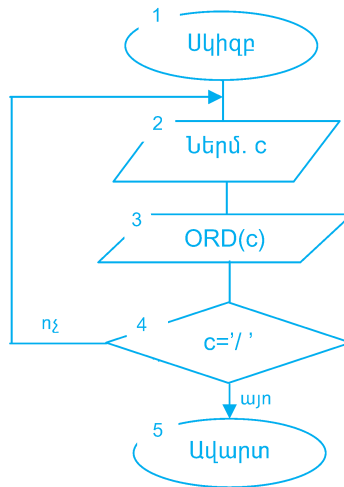
ցիկլի մարմին

UNTIL տրամաբանական արտահայտություն;

որտեղ *REPEAT*-ը և *UNTIL*-ը առանցքային բառեր են:

REPEAT և *UNTIL* բառերի միջև ներառվող *ցիկլի մարմինը* կազմվում է մեկ կամ մի քանի օպերատորներով, որոնք կրկնվելով իրագործվում են այնքան, քանի դեռ *UNTIL*-ի տրամաբանական արտահայտությունն ունի *FALSE* արժեք: Այս օպերատորի առանձնահատկությունն այն է, որ ցիկլի մարմինը, անկախ *UNTIL*-ի *տրամաբանական արտահայտության* արժեքից, գոնե մեկ անգամ իրագործվում է:

Նախորդ խնդրի բոկ-սխեման այժմ կառուցենք հետպայմանով ցիկլի օպերատորի միջոցով իրագործելու նպատակով.



Նկ. 1.13. Հեղապայմանով ցիկլային ալգորիթմի օրինակ

Գրենք համապատասխան ծրագիրը.

```
PROGRAM Het_Cikl;
VAR c:CHAR;
BEGIN
  REPEAT
    READLN(c);
    WRITELN('Ներմուծված պայմանանշանի կողքը
    հավասար է', ORD(c))
  UNTIL c='/'
END.
```

Ի տարբերություն նույն խնդրի նախապայմանով ցիկլի օգնությամբ լուծված տարբերակի, այստեղ ցիկլի ավարտից առաջ ներմուծված ‘/’ պայմանանշանի կողք ևս կարտածվի էկրանին:

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ Ցիկլի ցանկացած օպերատորի աշխատանքը կարելի է վաղաժամկետ ավարտել՝ դրա մարմնում **BREAK** օպերատորի կիրառմամբ:
- ◆ Պարամետրով ցիկլի պարամետրը ցանկացած կարգային փոփոխական է:
- ◆ Եթե ցիկլի պարամետրի փոփոխման քայլը 1 կամ -1 չէ, կամ եթե այն կարգային փոփոխ չէ (օրինակ՝ իրական թիվ է), ապա պարամետրով ցիկլի փոխարեն պետք է կիրառել նախապայմանով կամ հետպայմանով ցիկլի օպերատորներից որևէ մեկը:
- ◆ Հետպայմանով ցիկլի մարմինը, եթե նույնիսկ մեկից ավելի օպերատորներ է ներառում, կարիք չկա առնելու **BEGIN** և **END** բառերի միջև, քանի որ **REPEAT**-ը այս դեպքում հանդիսանում է նաև ցիկլի մարմնի սկիզբ, իսկ **UNTIL**-ը՝ վերջ:



1. Բերեք ցիկլային բնույթ կրող որևէ գործընթացի օրինակ:
2. Պարամետրով ցիկլի քանի՞ օպերատոր գիտեք:
3. Հնարավո՞ր է, որ նախապայմանով ցիկլի մարմինը ոչ մի անգամ չիրագործվի. եթե այո՝ ո՞ր դեպքում:
4. Հետպայմանով ցիկլի մարմինն ամենաքիչը քանի՞ անգամ է կատարվում:
5. Ցիկլի բնականոն ընթացքն ընդհատող ի՞նչ օպերատոր գիտեք:
6. Հաշվել և արտածել [1;15] միջակայքի ամբողջ թվերի արտադրյալը:
7. Արտածել այն հաջորդական 10 թվերը, որոնցից առաջինը հավասար է 86-ի, իսկ մնացածներից յուրաքանչյուրն իր նախորդից փոքր է 3-ով:
8. p տրամաբանական փոփոխականին վերագրել *true* արժեքը, եթե տրված n ($n > 1$) թիվը պարզ է, հակառակ դեպքում *false*: Արտածել p փոփոխականի արժեքը:
9. Հաշվել և արտածել այն երկնիշ թվերի գումարը, որոնք բազմապատիկ են 3-ին:
10. Կազմել հավելված 3-ի այս թեմային առնչվող խնդիրների լուծման բլոկ-սխեմաներն ու ծրագրերը:

§ 1.13 ՄԻԱԶԱՓ ԶԱՆԳՎԱԾՆԵՐ

Ծրագրավորման մեջ, բացի պարզ տիպերից, կիրառում են նաև, այսպես կոչված, **կառուցվածքային տիպեր**: Կառուցվածքային տիպի փոփոխականը կամ հաստատունը մի քանի տարրերից է բաղկացած:

Կառուցվածքային տիպերից նախ կուսումնասիրենք **զանգվածները**: Զանգվածի բաղադրիչ տարրերը միևնույն տիպի են: Յուրաքանչյուր տարր հերթական համար՝ **ինդեքս** ունի, որի միջոցով կարելի է դիմել դրան: Եթե զանգվածի տարրին դիմելիս միայն մեկ ինդեքս է կիրառվում, ապա այդ զանգվածն անվանում են **միաչափ**: Միաչափ զանգվածները հայտարարում են հետևյալ կերպ.

VAR ինդեքսի.ֆիկսարր : ARRAY[ինդեքսի սարորին եզր. . . ինդեքսի վերին եզր] OF տիպ;

որտեղ *ARRAY, OF* բառերն առանցքային բառեր են, *ինդեքսի սարորին* և *վերին եզրերը*՝ կարգային տիպի մեծություններ են (սովորաբար կիրառվում է *միջակայքային տիպը*): Օրինակ՝

```
a:ARRAY[1..10] OF INTEGER;
x:ARRAY[-2..5] OF CHAR;
y:ARRAY[FALSE..TRUE] OF REAL;
```

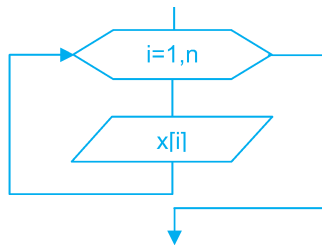
Զանգվածի հայտարարման մեջ *OF*-ից հետո գրված տիպով բնորոշում են զանգվածի տարրերը: Օրինակ՝ որպես վերը հայտարարված *a* զանգվածի տարրեր կարող են հանդես գալ հետևյալ թվերը՝ 8, -2, 0, 9, -5, 3, 3, 4, 100, -100, որտեղ 8-ը զանգվածի առաջին տարրն է կամ, որ նույնն է, *a[1]*-ը, -2-ը երկրորդ տարրը, կամ՝ *a[2]*-ը, ... -100-ը զանգվածի 10-րդ տարրը՝ *a[10]*:

Ինչպես երևում է օրինակից, տարրի ինդեքսը (հերթական համարը) գրվում է բառակուսի փակագծերի *[]* մեջ:

Զանգվածը կարելի է հայտարարել նաև, օրինակ, հետևյալ կերպ՝

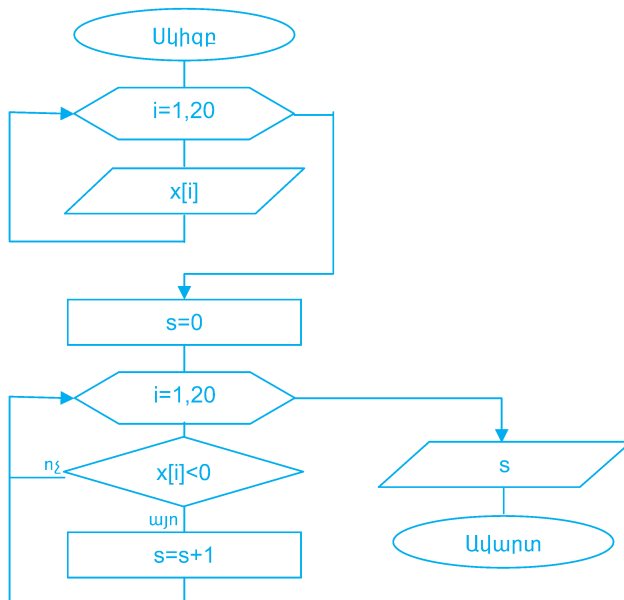
```
TYPE zangvac=ARRAY[1..10] OF REAL;
VAR x:zangvac;
```

Զանգվածների ներմուծումը (արտածումը) կատարվում է ցիկլի միջոցով, որտեղ հերթով ներմուծվում (արտածվում) են *x[1]*, *x[2]*, ..., *x[n]* տարրերի արժեքները:



Նկ. 1.14. *n* տարր պարունակող միաչափ զանգվածի տարրերի ներմուծման (արտածման) գործընթացի օրինակ

Դիտարկենք հետևյալ խնդիրը. *հաշվել իրական տիպի 20 տարր պարունակող միաչափ զանգվածի բացասական տարրերի քանակը:*



Նկ. 1.15. Չանգվածի բացասական տարրերի քանակի հաշվման ալգորիթմ

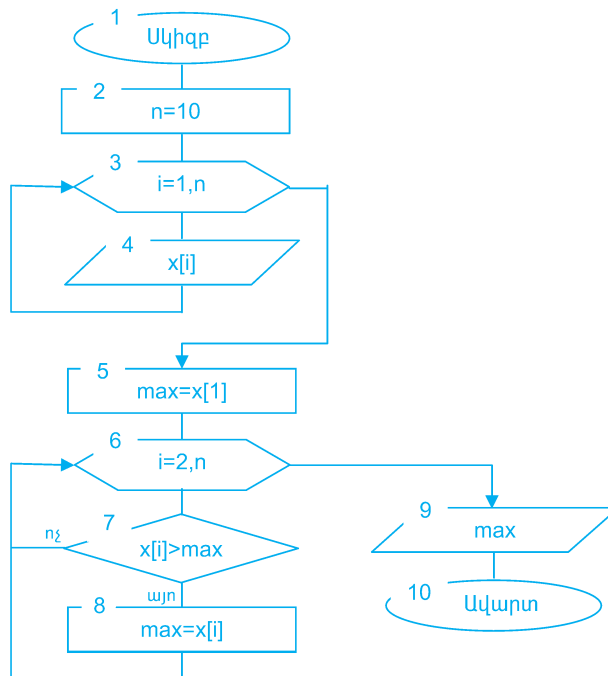
Կազմենք խնդրի լուծման ծրագիրը.

```

PROGRAM Bac_Qanak;
TYPE vektor=ARRAY[1..20] OF REAL;
VAR x:vektor; {զանգվածի հայտարարումը}
    i,s:BYTE; {i-ն ինդեքսի և s-ը պահանջվող քանակի համար է}
BEGIN
  FOR i:=1 TO 20 DO {1}
    BEGIN WRITE('x[',i,']= '); READ(x[i]); {2}
    END; {3}
  s:=0; {4}
  FOR i:=1 TO 20 DO {5}
    IF x[i]<0 THEN s:=s+1; {6}
  WRITELN('զանգվածի բաց. տարրերի քանակը=',s) {7}
END.
  
```

{1}-ից {3} տողերում 20 իրական թվեր են ներմուծվում, որոնք հաջորդաբար փոխանցվում են x զանգվածի տարրերին: Պահանջվող տարրերի քանակը հաշվելու համար s -ը {4} տողում սկզբնաբեքավորվում է նախնական 0 արժեքով, այնուհետև {5}-{6}-ում հերթով ստուգվում են զանգվածի բոլոր տարրերը, և եթե բացասական (<0) տարրեր կան, դրանց քանակը հաշվարկվում է $s:=s+1$ հրամանով: {7}-րդ տողում արտածվում է ստացված քանակը:

Հետևյալ խնդրի միջոցով որոշենք իրական տիպի 10 տարր պարունակող միաչափ զանգվածի մեծագույն տարրի արժեքը:



Նկ. 1.16. Չանգվածի մեծագույն արժեքի որոշման ալգորիթմ

Չանգվածի տարրերը ներմուծելուց հետո 5-րդ բլոկով կատարվել է $max=x[1]$ վերագրումը (կարելի է ասել՝ ենթադրվել է, թե մեծագույն տարրը $x[1]$ -ն է՝ զանգվածի առաջին տարրը), այնուհետև 6, 7 և 8-րդ բլոկներով ցիկլի միջոցով հաջորդաբար մնացած տարրերը համեմատվել են max -ի մեջ պահպանված արժեքի հետ: Եթե առավել մեծ արժեքով $x[i]$ տարր է հայտնաբերվել, ապա 8-րդ բլոկով max -ի արժեքը փոխարինվել է այդ առավել մեծ արժեքով: Յիկլի ավարտին max -ի մեջ կլինի փնտրված մեծագույն արժեքը:

Կազմենք ծրագիրը.

```

PROGRAM maxx;
CONST n=10;
VAR x:ARRAY[1..n] OF REAL;
    i:BYTE; max:REAL;
BEGIN
    FOR i:=1 TO n DO READ(x[i]);
    max:=x[1];
    FOR i:=2 TO n DO
        IF x[i]>max THEN max:=x[i];
        WRITELN('max=',max:6:2)
    END.
  
```

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ **Չանգվածի փարբերը համակարգչի հիշողությունում անընդմեջ հաջորդական փարածք են զբաղեցնում:**



1. Ի՞նչ է զանգվածը:
2. Ո՞ր զանգվածներն են անվանում միաչափ:
3. Կազմել հավելված 3-ի այս թեմային առնչվող խնդիրների լուծման բլոկ-սխեմաներն ու ծրագրերը:

§ 1.14 ԵՐԿՉԱՓ ՉԱՆԳՎԱԾՆԵՐ

Երկչափ զանգվածը բնութագրվում է նախ և առաջ այն բանով, որ դրա յուրաքանչյուր տարրին դիմելու համար անհրաժեշտ է երկու ինդեքս կիրառել: Երկչափ զանգվածն ակնառու պատկերացնելու համար դիտենք դասարանի նստարանների շարքերը: Ենթադրենք, դրանք դասավորված են 4 շարքով, իսկ յուրաքանչյուր շարքում 5-ական նստարաններ կան. համարակալները դրանք այնպես, որպեսզի այդ համարներով միարժեքորեն որոշվի նստարանի գտնվելու շարքը և շարքում ունեցած դիրքի համարը:

$S[1,1]$	$S[1,2]$	$S[1,3]$	$S[1,4]$
$S[2,1]$	$S[2,2]$	$S[2,3]$	$S[2,4]$
$S[3,1]$	$S[3,2]$	$S[3,3]$	$S[3,4]$
$S[4,1]$	$S[4,2]$	$S[4,3]$	$S[4,4]$
$S[5,1]$	$S[5,2]$	$S[5,3]$	$S[5,4]$

Ինչպես տեսնում եք, բերված օրինակում շարքը բնորոշող համարը քառակուսի փակագծերի մեջ առնված թվերից երկրորդն է, իսկ առաջինը՝ տվյալ շարքում նստարանի հերթական համարը: Նման եղանակով կարգավորված տվյալների համախումբն անվանում են **երկչափ զանգված**: Երկչափ զանգվածի տարրի **առաջին ինդեքսը** համարում են տարրի գտնվելու **դրոշի**, իսկ երկրորդը՝ **սյան** համարը: Նույն սկզբունքով կարելի է սահմանել նաև բազմաչափ զանգվածները:

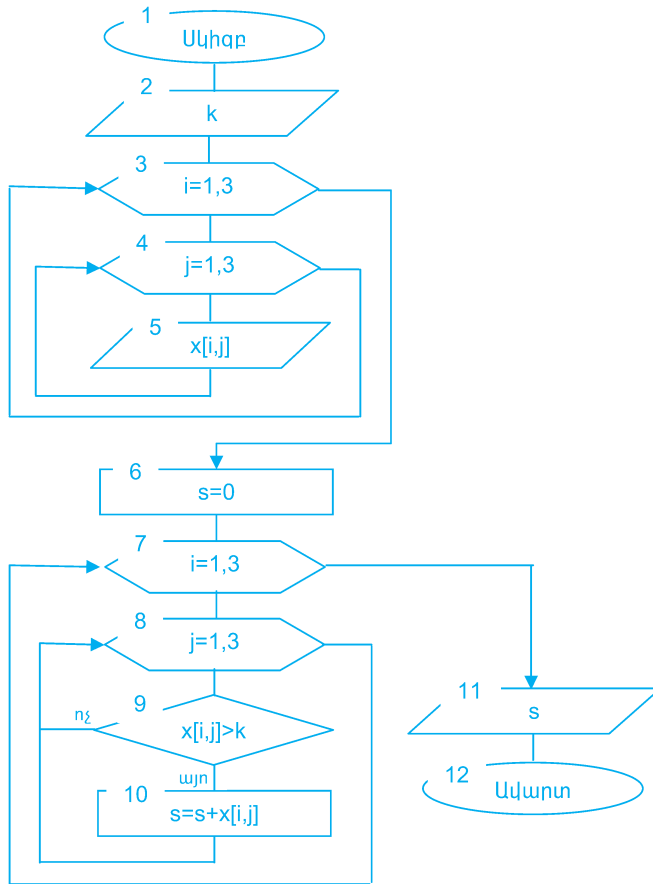
Երկչափ զանգվածներ հայտարարելիս պետք է նշել դրա երկու ինդեքսների փոփոխման միջակայքերը, օրինակ՝

$VAR x:ARRAY[1..10,1..20] OF REAL;$

որտեղ փակագծերում 1..10-ը ցույց է տալիս առաջին ինդեքսի (տողի համարի), իսկ 1..20-ը՝ երկրորդ ինդեքսի (սյան համարի) փոփոխման տիրույթը:

Երկչափ զանգվածների հետ կապված աշխատանքին ավելի մոտիկից ծանոթանալու նպատակով մի քանի խնդիր լուծենք:

Խնդիր. տրված է 3×3 (3 տող և 3 սյուն) ամբողջ տիպի տարրեր պարունակող երկչափ զանգված: Հաշվել զանգվածի տրված k ամբողջ բվից մեծ արժեք ունեցող տարրերի գումարը:



Նկ. 1.17. Երկչափ զանգվածում գումարի հաշվման ալգորիթի

Ինչպես երևում է բլոկ-սխեմայից, երկչափ զանգված ներմուծելու համար մեկը {4} մյուսի {3} մեջ ներդրված ցիկլեր են կիրառվել: **Ներդրված ցիկլերն** աշխատում են հետևյալ կերպ. նախ արտաքին {3} ցիկլի i պարամետրը ստանում է իր սկզբնական 1 արժեքը, և ղեկավարումը տրվում է ներդրված {4} ցիկլին, վերջինս ցիկլի սովորական, մեզ արդեն հայտնի սխեմայով է աշխատում, այսինքն՝ $i=1$ արժեքի դեպքում j -ն փոփոխվելով 1 -ից 3 ՝ ներմուծվում են i -րդ (այս պահին՝ 1 -ին) տողի տարրերը, այնուհետև ղեկավարումը կրկին տրվում է {3} բլոկին, որտեղ i -ն աճելով ստանում է 2 արժեքը, և ամեն ինչ ընթանում է այնպես, ինչպես $i=1$ արժեքի դեպքում, այսինքն՝ այժմ ներմուծվում են 2 -րդ տողի տարրերը: Նույնը կատարվում է նաև $i=3$ -ի դեպքում: Աշխատանքի այսպիսի ընթացքը հատկանշական է ցանկացած ներդրված ցիկլի համար:

Այժմ կազմենք գրված բլոկ-սխեմային համապատասխանող ծրագիրը.

```

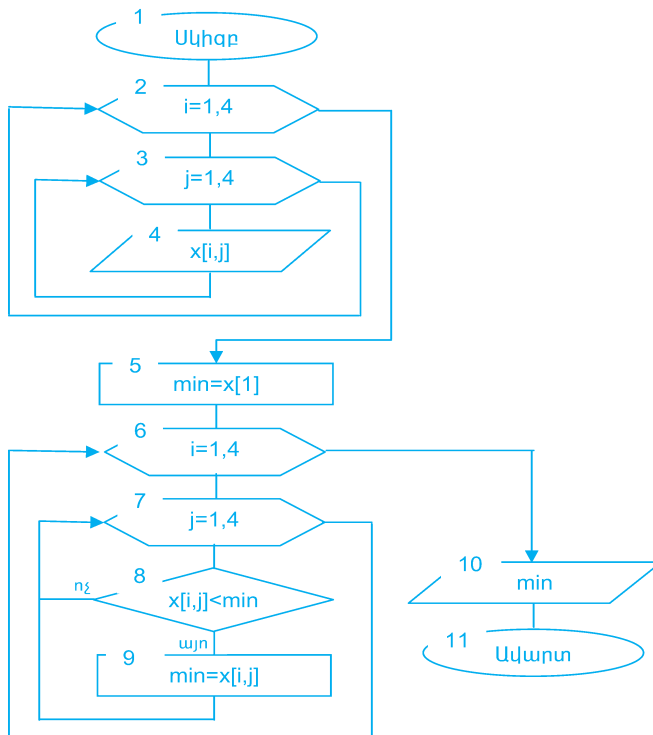
PROGRAM Matrici_Khndir;
CONST n=3;
TYPE matric=ARRAY[1..n,1..n] OF INTEGER;
VAR i,j:BYTE; k,s:INTEGER; x:matric;
BEGIN WRITE('k= ');READ(k);
      FOR i:=1 TO n DO                                {1}
        FOR j:=1 TO n DO                              {2}
          READ(x[i,j]);                               {3}
        s:=0;
        FOR i:=1 TO n DO
          FOR j:=1 TO n DO
            IF x[i,j]>k THEN s:=s+x[i,j];
            WRITELN('k-ից մեծ տարրերի գումարը= ',s:10)
          END.

```

Ծրագրում {1} և {2} մեկնաբանությանը ցիկլերի միջև *BEGIN*-ի անհրաժեշտություն չկա, քանի որ {1} ցիկլում մեկ օպերատոր կա՝ {2}-ը, իսկ {3}-ը {2}-ի միակ կրկնվող օպերատորն է:

Կազմենք հետևյալ խնդրի լուծման բլոկ-սխեման ու ծրագիրը ևս. *տրված է 4x4 հրական տիպի տարրեր պարունակող երկչափ զանգված: Հաշվել և արտածել զանգվածի փոքրագույն տարրի արժեքը:*

Նախ կազմենք խնդրի լուծման բլոկ-սխեման:



Նկ. 1.18. Երկչափ զանգվածի փոքրագույն տարրի որոշման ալգորիթմ

Խնդրի լուծման ալգորիթմը նման է միաչափ զանգվածի մեծագույն տարրը որոշելու ալգորիթմին: Նախ min փոփոխականի մեջ պահվել է զանգվածի տարրերից առաջինը (ընդ որում՝ կարևոր չէ, թե որ տարրի արժեքն այս պահին կդիտվի որպես փոքրագույն), որից հետո 6 և 7-րդ բլոկներով կազմավորված ներդրված ցիկլերի միջոցով ենթադրյալ փոքրագույնի (min) հետ համեմատվել են զանգվածի մնացած բոլոր տարրերը և արժեքով առավել փոքր տարրը 9-րդ բլոկում վերագրվել է min -ին: Վերջում min -ի մեջ կմնա զանգվածի փոքրագույն տարրը, որի արժեքն արտածվել է 10-րդ բլոկով:

Կազմենք ծրագիրը.

```
PROGRAM Min_Matric;
CONST n=4;
VAR x:ARRAY[1..n,1..n] OF REAL;
    i,j:BYTE; min:REAL;
BEGIN FOR i:=1 TO n DO
      FOR j:=1 TO n DO
        READ(x[i,j]);
        min:=x[1,1];
      FOR i:=1 TO n DO
        FOR j:=1 TO n DO
          IF x[i,j]<min THEN min:=x[i,j];
        WRITE('min= ',min:10:2)
      END.
```

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ **Չանգվածը, անկախ իր չափողականությունից, չի կարող 65520 բայթից ավելի հիշողություն զբաղեցնել:**



- Կարո՞ղ են երկչափ զանգվածի առաջին տողի տարրերը լինել սիմվոլային տիպի, իսկ մնացած տարրերը, օրինակ, ամբողջ տիպի:
- Եթե զանգվածի տարրը բնորոշվում է որպես $x[a,b,c]$, ապա զանգվածը
 - երկչափ է,
 - միաչափ է,
 - եռաչափ է,
 - $a * b * c$ չափի է:
- Կարո՞ղ է զանգվածի ինդեքսն իրական թիվ լինել:
- Կազմել հավելված 3-ի այս բեմային առնչվող խնդիրների լուծման բլոկ-սխեմաներն ու ծրագրերը:

§ 1.15 ԵՆՔԱԾՐԱԳԻՐ-ՊՐՈՑԵԴՈՒՐԱ

Ծրագրերը հաճախ այնպիսի հատվածներ են պարունակում, որոնք տարբեր տվյալների համար միևնույն ալգորիթմական գործընթացն են իրականացնում: Նման հատվածները սովորաբար ձևակերպվում են որպես համեմատաբար ինքնուրույն ծրագրային միավորներ, որոնք կոչվում են **ենթածրագրեր**: Այսպիսով, ծրագրից «առանձնացվում են» իմաստով անկախ, ինքնուրույն կառույցները, որոնք, գրվելով միայն մեկ անգամ, հնարավոր է բազմիցս աշխատեցնել նախնական տարբեր տվյալների համար: Նման կառույցները թույլատրում են ոչ միայն հիմնական ծրագրի ծավալը մեծապես կրճատել, այլև ծրագիրը դարձնել առավել ընթեռնելի, հասկանալի, հեշտացնել ծրագրում առկա սխալների հայտնաբերումն ու ծրագրի կարգաբերումը:

Կարելի է ասել, որ ենթածրագրերի կիրառմամբ կառուցվածքային ծրագրավորումը մի նոր, առավել բարձր փուլ թևակոխեց:

Նկ. 1.19-ում բերված ընդհանրական սխեմայի օրինակով հետևենք ենթածրագրի կիրառման գործընթացին: Ծրագրի նախնական մարմինը միևնույն Ալգորիթմ 1-ի կիրառման երեք տեղամասեր է պարունակում (նկ 1.19 ա): Այդ տեղամասերի փոխարեն 1.19 բ) տարբերակում հիմնական ծրագրից *Ալգորիթմ 1-ի* իրագործման մասն առանձնացվել է որպես ենթածրագիր, իսկ հիմնական ծրագրում *Ալգորիթմ 1-ին* համապատասխանող մասերը փոխարինվել են այդ **ենթածրագրի կանչերով**: Ակնհայտ է, որ ենթածրագրի կիրառման արդյունքում ծրագրի հիմնական մարմինը ծավալով կրճատվել է:

Պասկալում կիրառվում են երկու տիպի ենթածրագրեր՝ **պրոցեդուրաներ** և **ֆունկցիաներ**:

Պրոցեդուրաների կառուցվածքը նման է հիմնական ծրագրի կառուցվածքին.

Պրոցեդուրայի վերնագիր:

Պրոցեդուրայում օգտագործվող մեծությունների նկարագրություններ:

BEGIN

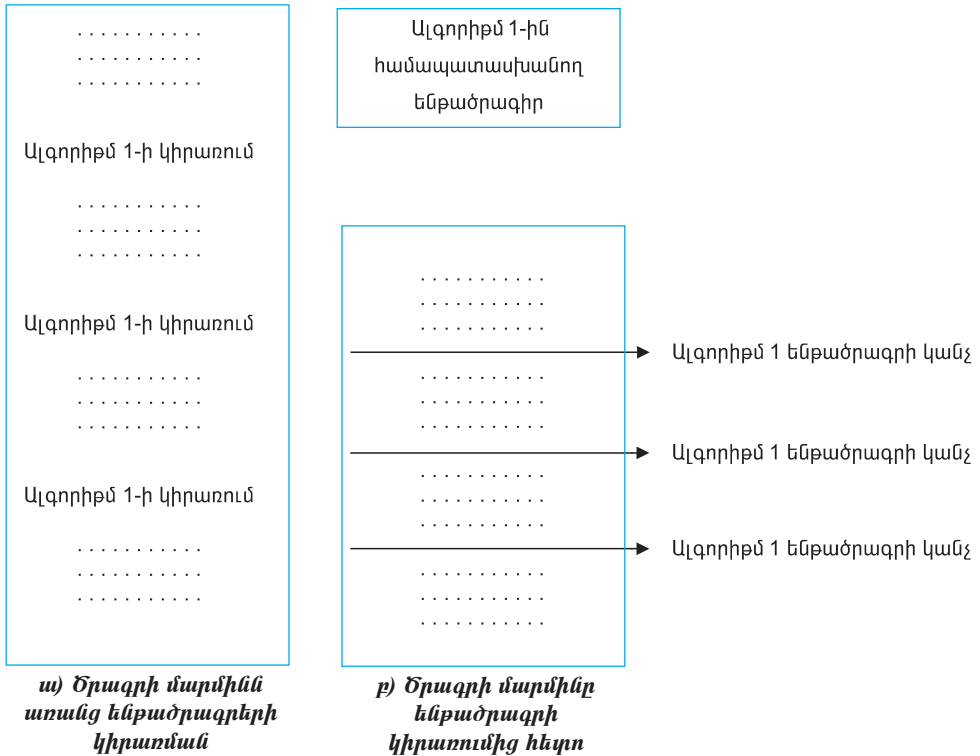
Պրոցեդուրայի մարմին

END;

Պրոցեդուրայի վերնագիրը, ի տարբերություն ծրագրի վերնագրի, պարտադիր է: Այն ունի հետևյալ ընդհանուր տեսքը.

PROCEDURE պրոցեդուրայի անուն (պարամետրերի նկարագրություն):

որտեղ *PROCEDURE*-ն առանցքային բառ է, *պրոցեդուրայի անունը*՝ ցանկացած իդենտիֆիկատոր: Այստեղ իր էական նշանակությամբ առանձնանում է փակագծերի մեջ առնված *պարամետրերի նկարագրության* մասը: Բանն այն է, որ այստեղ են իրականացվում դրսից պրոցեդուրային փոխանցվող և պրոցեդուրայից դուրս առաքվող մեծությունների նկարագրությունները: Այլ խոսքով, պրոցեդուրայի վերնագրից կարելի է տեսնել, թե պրոցեդուրան դրսից «ի՞նչ է ստանում» և դուրս «ի՞նչ է ուղարկում»:



Նկ. 1.19. Ենթածրագրի կիրառման սխեմա

Ընդհանուր առմամբ ենթածրագրի վերնագրի մեջ ներառված բոլոր մեծություններն անվանում են **ֆորմալ (չեական) պարամետրեր**: Այդ պարամետրերի **ֆորմալ** կոչվելու պատճառն այն է, որ դրանք ընդհանրական բնույթ կրող պարամետրեր են, որոնք ենթածրագրի ամեն կոնկրետ կանչի դեպքում նոր **փաստացի արժեքներ** են ստանում: Սա նման է հետևյալ օրինակին՝ ունենք 200մլ տարողությամբ բաժակ (**ֆորմալ պարամետր**), որը կարող է թեյ կամ կաթ (**փաստացի պարամետրեր**) պարունակել՝ նայած թե տվյալ պահին դրա մեջ ինչ ենք լցնում:

Ըստ **ֆորմալ պարամետրի կիրառման նպատակի**՝ տարբերում են **արժեք պարամետր** և **փոփոխական պարամետր** հասկացությունները:

Արժեք պարամետրի միջոցով ենթածրագիրը **դրսից** (իրեն կանչող ծրագրային մոդուլից) արժեքներ է ստանում:

Փոփոխական պարամետրերը ոչ միայն կարող են արժեք պարամետրի պես դրսից արժեքներ ընդունել, այլև ենթածրագրից արժեքներ դուրս ուղարկել: Որպեսզի **Պասկալի** կոմպիլյատորն արժեք պարամետրը փոփոխական պարամետրից տարբերի՝ ընդունված է դրանք նկարագրել տարբեր ձևերով. **փոփոխական պարամետրի** նկարագրումը սկսվում է **VAR** առանցքային բառով, իսկ **արժեք պարամետրը** նկարագրվում է առանց դրա: Օրինակ՝

```
PROCEDURE aa(b:INTEGER; VAR k:CHAR; VAR d:REAL; c:BYTE);
```


Այստեղ k և d պարամետրերը փոփոխական պարամետրեր են, իսկ b -ն և c -ն՝ արժեք պարամետրեր: Ընդ որում՝ նույնատիպ փոփոխական պարամետրերը կարելի է համախմբել միևնույն VAR -ի տակ: Օրինակ՝

PROCEDURE bb(VAR r:REAL; VAR d:REAL);

և

PROCEDURE bb(VAR r, d:REAL);

վերնագրերը համարժեք են:

Շարունակենք ուսումնասիրել պրոցեդուրայի ընդհանուր կառուցվածքը: Պրոցեդուրայի վերնագրից հետո բերվում են այն մեծությունների նկարագրությունները, որոնք, պրոցեդուրայի մարմնում օգտագործվելով, չեն ընդգրկվել պրոցեդուրայի վերնագրում (պարամետրեր չեն հանդիսանում): Նման մեծություններն անվանում են **լոկալ (տեղային)** մեծություններ: Բանն այն է, որ լոկալ մեծությունները ենթաձրագրի մարմնից դուրս ճանաչելի չեն: Ի տարբերություն լոկալ մեծությունների, ձրագրի սկզբում (ենթաձրագրերից դուրս) նկարագրված մեծությունները կոչվում են **գլոբալ**, քանի որ դրանք կարելի է օգտագործել թե՛ ձրագրի հիմնական մարմնում և թե՛ ենթաձրագրերի մեջ:

Եթե միևնույն անունով թե՛ գլոբալ և թե՛ լոկալ մեծություններ կան, ապա լեզվի կոմպիլյատորը տարբերում է դրանք՝ ենթաձրագրում կիրառելով համապատասխանաբար լոկալ, իսկ ձրագրում՝ գլոբալ մեծությունները:

Պրոցեդուրայի մարմինը բլոկ, այսինքն՝ *BEGIN* և *END* առանցքային բառերի միջև առնված օպերատորների հաջորդականություն է, որտեղ *END*-ն ավարտվում է կետ-ստորակետով (*;*):

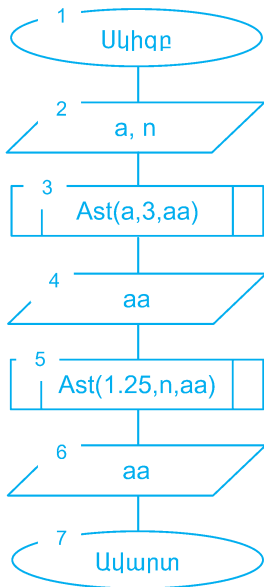
Կազմենք հետևյալ խնդրի բլոկ-սխեման ու ձրագիրը. *հաշվել ցանկացած a թվի ($a \neq 0$) n աստիճանը, որտեղ՝*

ա) a -ն ցանկացած թիվ է, $n=3$,

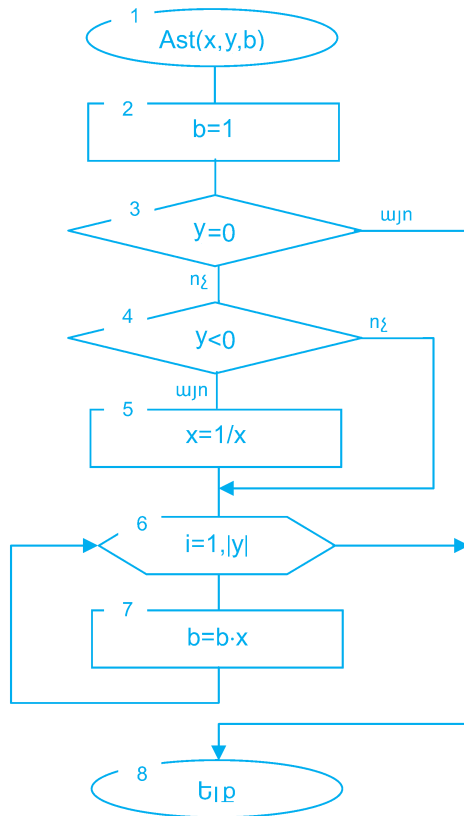
բ) $a=1.25$, n -ը ցանկացած ամբողջ թիվ է:

Խնդիրը լուծենք պրոցեդուրայի կիրառմամբ, որն ըստ իրեն փոխանցված x և y պարամետրերի՝ հաշվում է x -ի y աստիճանը:

Ինչպես տեսնում եք՝ խնդրի լուծման բլոկ-սխեման (նկ. 1.20) երկու (*ա*) և *բ*) մասերից է բաղկացած. *ա*)-ն գլխավոր կամ, այսպես ասած, **դեկավարող բլոկ-սխեման** է, իսկ *բ*)-ն՝ *Ast(x,y,b)* պրոցեդուրային համապատասխանող բլոկ-սխեման: *ա*) բլոկ-սխեմայում 3-րդ և 5-րդ բլոկների միջոցով կանչ է կատարվել *Ast* պրոցեդուրային, ընդ որում՝ առաջին կանչի դեպքում (3-րդ բլոկ) պրոցեդուրային փոխանցվել են երկու արժեքներ՝ ներմուծված a -ն, որը պրոցեդուրան կվերցնի որպես x -ի արժեք և 3-ը, որը պրոցեդուրան կընդունի որպես y -ի արժեք, իսկ aa -ն այս դեպքում նախատեսված է պրոցեդուրայից փոխանցվելիք a^3 -ի արժեքը վերցնելու համար: Ստացված արժեքը 4-րդ բլոկով արտաձեղուց հետո նորից (5-րդ բլոկ) կանչ է կատարվել *Ast(x,y,b)* պրոցեդուրային, սակայն այժմ x -ին փոխանցվել է 1.25-ը, իսկ y -ին՝ ներմուծված n -ի արժեքը: Այժմ aa -ի մեջ կստանանք 1.25ⁿ-ի արժեքը: Ուսումնասիրենք 1.20 *բ*) բլոկ-սխեման. այստեղ նախ ստուգվել է որպես աստիճան կիրառվող y -ի արժեքը. եթե հավասար է 0-ի, ապա 2-րդ բլոկում ստացված $b=1$ պատասխանն արդեն հանդիսանում է



ա) ղեկավարող բլոկ-սխեմա



բ) պրոցեդուրայի բլոկ-սխեմա

Նկ. 1.20. a բլի n աստիճանը հաշվելու ալգորիթմ պրոցեդուրայի կիրառմամբ

x^y -ի արժեքը, հակառակ դեպքում ստուգվել է $y < 0$ պայմանը, և եթե y -ը բացասական թիվ է, ապա, օգտվելով $x^{-y} = (1/x)^y$ հավասարությունից, 6-րդ և 7-րդ բլոկներով հաշվվել է անհրաժեշտ աստիճանն ու ավարտվել ենթաձրագիրը:

Այժմ բերենք նկարագրված բլոկ-սխեմաներին համապատասխանող ծրագիրը.

```

PROGRAM Proc_1;
VAR a,aa:REAL;n:INTEGER;                                {0}
PROCEDURE Ast(x:REAL;y:INTEGER;VAR b:REAL);            {1}
VAR i:WORD;                                             {2}
BEGIN                                                  {3}
    b:=1;
    IF y<>0 THEN
    BEGIN
        IF y<0 THEN x:=1/x;
        FOR i:=1 TO ABS(y) DO b:=b * x
    END
END
  
```

<i>END;</i>	{4}
<i>BEGIN WRITE('a= '); READ(a);</i>	{5}
<i>WRITE('n= '); READ(n);</i>	
<i>Ast(a,3,aa);</i>	{6}
<i>WRITELN(a, '-ի խորանարդը=' ,aa:7:2);</i>	
<i>Ast(1.25,n,aa);</i>	{7}
<i>WRITELN(1.25, '-ի',n, '-րդ աստիճանը=' ,aa:7:2);</i>	
<i>END.</i>	{8}

Ծրագրի {0} տողում հայտարարվել են գլոբալ փոփոխականները, սրանք այն փոփոխականներն են, որոնք օգտագործվել են *ա*) բլոկ-սխեմայում: {1} տողից մինչև {4}-ն ընկած մասում նկարագրված է *Ast* պրոցեդուրան:

{1} տողում գրված է պրոցեդուրայի վերնագիրը, ըստ որի պրոցեդուրան ունի մուտքային երկու՝ *x* և *y* (արժեք) և ելքային մեկ *b* (փոփոխական) պարամետրեր: {2}-րդ տողում նկարագրված է պրոցեդուրայի միակ լոկալ փոփոխականը՝ *i*-ն: {3}-ից {4}-ն ընկած ծրագրային հատվածի օգնությամբ հաշվարկվել և ելքային *b* պարամետրի մեջ պահպանվել է պահանջվող x^y արժեքը:

{5}-րդ տողով սկսվել է ծրագրի հիմնական մասը կազմող հատվածը: {6}-րդ տողում *Ast* պրոցեդուրան կանչվել է առաջին անգամ: Այս անգամ դրան փոխանցվել է *a*-ի արժեքը (պրոցեդուրան այն ստացել է *x*-ի մեջ) և 3-ը (այս արժեքը պրոցեդուրան ընդունել է *y*-ի մեջ), իսկ *aa* փոփոխականը (սրան պրոցեդուրայում համապատասխանել է *b* փոփոխական պարամետրը) նախատեսված է պրոցեդուրայից վերադարձվող արժեքն ընդունելու համար: Երկրորդ անգամ պրոցեդուրան կանչվել է {7}-րդ տողում, այս անգամ 1.25-ն է փոխանցվել պրոցեդուրային և ընդունվել *x*-ի մեջ, *n*-ը, որն ընդունվել է *y*-ի մեջ, իսկ *aa*-ի մեջ նորից ստացվել է պրոցեդուրայից վերադարձվող նոր արժեքը:

ՕՉՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ **Պրոցեդուրան կարող է բազմաթիվ արժեքներ վերադարձնել, այլ խոսքով՝ կարող է այնքան փոփոխական պարամետրեր պարունակել, որքան անհրաժեշտ է:**
- ◆ **Ենթածրագրերը նույնպես կարող են այլ ենթածրագրերի կանչեր պարունակել:**
- ◆ **Պասկալում պահպանվել և խորացվել է ծրագրավորման վերից վար սկզբունքը. այսպիսով, ավելի ներքևում հայտարարված ենթածրագիրը կարող է իր մարմնից վերև տեղադրված այլ ենթածրագրին կանչ ուղարկել, իսկ վերինը՝ ներքինին՝ ոչ:**
- ◆ **Ենթածրագրի կանչում ներկայացված պարամետրերը կոչվում են փաստացի պարամետրեր:**
- ◆ **Մինևույն ենթածրագրին համապարասխանող փաստացի և ֆորմալ պարամետրերը ոչ միայն պեպք է քանակներով համընկնեն, այլև յուրաքանչյուր փաստացի պարամետր իր տիպով պեպք է համընկնի ենթածրագրի վերնագրում միևնույն դիրքում դրան համապարասխանող ֆորմալ պարամետրի տիպի հեպ:**



1. Ի՞նչ է ենթածրագիրը, ո՞րն է այն կիրառելու հիմնական նպատակը:
2. Պասկայում քանի՞ տիպի ենթածրագրեր կան, ինչպե՞ս են կոչվում դրանք:
3. Որո՞նք են կոչվում գլոբալ փոփոխականներ և ծրագրի ո՞ր մասում են դրանք հայտարարվում:
4. Ո՞ր փոփոխականներն են կոչվում լոկալ:
5. Ի՞նչ է ֆորմալ պարամետրը և ինչո՞ւ է այն կոչվում ֆորմալ:
6. Ո՞րն է արժեք պարամետրի իմաստը:
7. Ինչպե՞ս են հայտարարում փոփոխական պարամետրը. ո՞րն է փոփոխական պարամետրի իմաստը:
8. Կազմել բլոկ-սխեմա և ծրագիր, որոնք պրոցեդուրայի միջոցով կհաշվեն և կարտածեն իրարից փարբեր *a*, *b* և *c* պարամետրերից
ա) մեծագույնի արժեքը,
բ) փոքրագույնի արժեքը:

§ 1.16 ԵՆԹԱԾՐԱԳԻՐ-ՖՈՒՆԿՑԻԱ

Պասկալ լեզվում կիրառվող ենթածրագրերից երկրորդը **ֆունկցիան** է: Ֆունկցիան ունի հետևյալ ընդհանուր կառուցվածքը.

Ֆունկցիայի վերնագիր:

Ֆունկցիայում օգտագործվող մեծությունների նկարագրություններ:

BEGIN

Ֆունկցիայի մարմին

END;

Ֆունկցիայի վերնագիրը, պրոցեդուրայի վերնագրի պես, նույնպես խիստ էական նշանակություն ունի և պարտադիր է: Այն ունի հետևյալ ընդհանուր տեսքը:

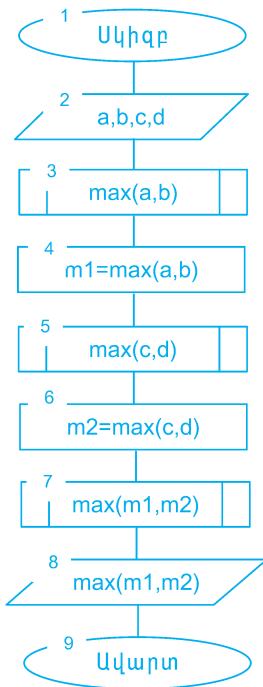
FUNCTION ֆունկցիայի անուն (ֆորմալ պարամետրեր): վերադարձվող արժեքի տիպ:

Ֆունկցիայի անունը ցանկացած իդենտիֆիկատոր է:

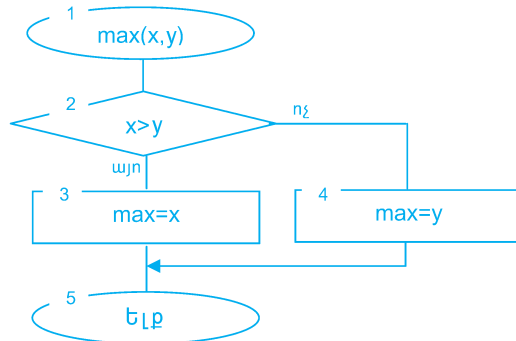
Ֆունկցիայի տարբերությունը պրոցեդուրայից առաջին հերթին վերնագրի տեսքի մեջ է: Այստեղ ֆորմալ պարամետրերի ցանկն ավարտող փակագծերից հետո դրվում է վերջակետ (:), որից հետո նկարագրվում է ֆունկցիայից վերադարձվող արժեքի տիպը: **Վերադարձվող արժեքը** կարող է լինել կարգային և իրական տիպերի: Մյուս տարբերությունն այն է, որ ֆունկցիայի մարմնում պետք է գոնե մեկ անգամ հանդիպի **ֆունկցիայի անվանն արժեք վերագրելու հրաման**:

Ֆունկցիայի կիրառմամբ լուծենք հետևյալ խնդիրը. **որոշել տրված *a*, *b*, *c* և *d* իրարից տարբեր արժեք ունեցող պարամետրերից մեծագույնի արժեքը՝ երկու պարամետրերից մեծագույնը հաշվող ֆունկցիայի կիրառմամբ:**

Նկ. 1.21-ում բերված է խնդրի լուծման բլոկ-սխեման. դիտարկենք այն:



ա) ղեկավարող բլոկ-սխեմա



բ) ֆունկցիայի բլոկ-սխեմա

Նկ. 1.21. Ֆունկցիայի կիրառմամբ 4 քվերից մեծագույնը որոշելու ալգորիթմ

Ղեկավարող (ա) բլոկ-սխեմայի 3-րդ բլոկով $\max(x,y)$ ֆունկցիան կանչվել է հաշվելու համար առաջին երկու՝ a և b պարամետրերից մեծագույնի արժեքը, որը 4-րդ բլոկում վերագրվել է $m1$ փոփոխականին: Այնուհետև 5-րդ բլոկով նորից կանչվել է $\max(x,y)$ ֆունկցիան, այս անգամ c և d պարամետրերից մեծագույնը հաշվելու համար, որը պահպանվել է $m2$ -ի մեջ (բլոկ 6): Այժմ խնդիրը լուծելու համար մնում է մեկ անգամ ևս կիրառել $\max(x,y)$ ֆունկցիան (բլոկ 7)՝ այս անգամ հաշվելու $m1$ և $m2$ պարամետրերից մեծագույնը, որն էլ 8-րդ բլոկով արտաձվել է: Նկ. 1.21 բ)-ում բերված է $\max(x,y)$ ֆունկցիայի բլոկ-սխեման, որը x և y պարամետրերից մեծագույնը վերագրում է ֆունկցիայի \max անվանն ու ավարտում աշխատանքը:

Գրենք բերված բլոկ-սխեմային համապատասխանող ծրագիրը.

```

PROGRAM Func_max;
VAR a,b,c,d,m1,m2:real;
    FUNCTION max(x,y:REAL):REAL; {1}
BEGIN
    IF x>y THEN max:=x ELSE max:=y
END; {2}
  
```

```

BEGIN                                                                 {3}
    WRITE('a= ');READ(a); WRITE('b= ');READ(b);
    WRITE('c= ');READ(c); WRITE('d= ');READ(d);
    m1:=max(a,b);                                                    {4}
    m2:=max(c,d);                                                    {5}
    WRITE(max(m1,m2):4:2)                                           {6}
END.

```

Ծրագրում {1}-ից {2} մակագրությամբ տողերում նկարագրվել է $\max(x,y)$ ֆունկցիան: Ֆունկցիայի մարմնում կիրառված $\max:=x$ և $\max:=y$ վերագրման օպերատորների շնորհիվ ֆունկցիան վերադարձնում է համապատասխանաբար x -ի կամ y -ի (սրանցից մեծի) արժեքը: {3}-ով սկսվում է ղեկավարող ծրագրի մարմինը: Պահանջվող փոփոխականների արժեքների ներմուծումից հետո {4} տողում կանչ է կատարվել \max ֆունկցիային՝ a և b փաստացի պարամետրերով. շնորհիվ $m1:=\max(a,b)$ վերագրման օպերատորի՝ ֆունկցիայից վերադարձված արժեքը պահվել է $m1$ -ում: {5} տողում c և d փաստացի պարամետրերով կանչված \max ֆունկցիան այս պարամետրերի մեծագույն արժեքն է վերադարձրել, որը պահպանվել է $m2$ -ի մեջ: Երրորդ անգամ \max -ը կանչվել է {6}-ում կիրառված $WRITE$ հրամանում՝ այս անգամ արդեն $m1$ և $m2$ փաստացի պարամետրերով. արդյունքում վերադարձված արժեքը պահանջված պատասխանն է, որն էլ արտածվել է:

Ինչպես նկատեցիք, ֆունկցիայի կանչը, ի տարբերություն պրոցեդուրայի կանչի, օգտագործվում է արտահայտությունների մեջ. անիմաստ է այն կիրառել պրոցեդուրայի կանչի պես, օրինակ, եթե {4} տողը գրվեր

$$\max(a,b); m1:=\max(a,b);$$

երկու հրամանների միջոցով, ապա սրանցից առաջինը ($\max(a,b)$) ժամանակի կորուստ կլիներ, քանի որ ֆունկցիայի վերադարձրած արժեքը չկիրառվելով՝ կկորչեր:

Պասկալում թույլատրվում է ենթածրագրի մարմնից նորից դիմել ընթացիկ ենթածրագրին. դա կոչվում է ենթածրագրի **ռեկուրսիվ կանչ**:

Ֆունկցիայի ռեկուրսիվ կիրառմամբ *հաշվենք $n!$ (n -ի ֆակտորիալի) արժեքը, որտեղ $n \geq 0$: Հիշեցնենք, որ*

$$n! = \begin{cases} 1, & \text{եթե } n = 0, \\ 1 \cdot 2 \cdot \dots \cdot (n-1) \cdot n, & \text{եթե } n > 0: \end{cases}$$

Կազմենք խնդրի լուծման բլոկ-սխեման (նկ. 1.22):

Ըստ նկ. 1.22-ում բերված բլոկ-սխեմայի՝ ծրագիրը կունենա հետևյալ տեսքը.

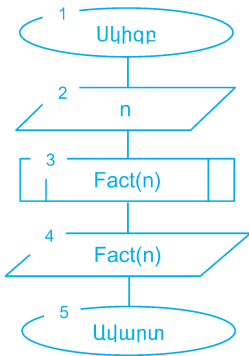
```

PROGRAM Factorial;
VAR n:BYTE;
    FUNCTION Fact(m:BYTE):LONGINT;
    BEGIN
        IF (m=0)OR(m=1) THEN Fact:=1
            ELSE Fact:=m * Fact(m-1)
        END;
    BEGIN

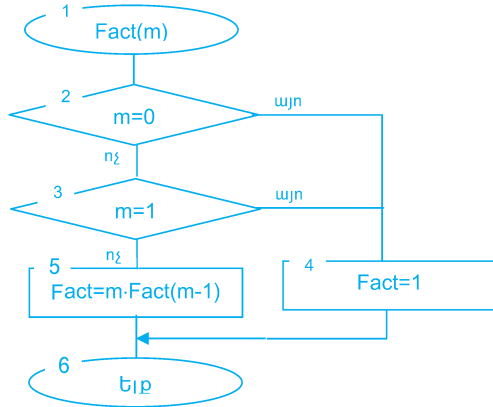
```

```

REPEAT
  READ (n)
  UNTIL (n >= 0);
  WRITE('n! = ', Fact(n))
END.
    
```



ա) դեկլարող բլոկ-սխեմա



բ) ֆունկցիայի բլոկ-սխեմա

Նկ. 1.22. Ռեկուրսիայի կիրառմամբ ֆակտորիալի հաշվման ալգորիթմ

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

◆ **Ենթածրագիր-ֆունկցիան նույնպես կարող է պրոցեդուրայի նման փոփոխական պարամետրերի միջոցով արժեքներ վերադարձնել:**



1. Ինչո՞վ է ֆունկցիան փարբերվում պրոցեդուրայից:
2. Ֆունկցիան իր անվան միջոցով քանի՞ արժեք կարող է վերադարձնել:
3. Եթե $\text{max}(a, b: \text{REAL}): \text{REAL}$; -ը ֆունկցիայի վերնագիրն է, ապա n ռ գրառումներն են ճիշտ.
 - ա) $\text{max}(a, b) := 7.8$;
 - բ) $\text{max} := 0.1$;
 - գ) $\text{max}(5, 6) := 0.25$;
 - դ) $\text{max} := 100$;

Կազմեք հեղեյալ խնդիրների լուծման բլոկ-սխեմաներն ու ծրագրերը.

4. Որոշել a, b, c և $4, 5, 6$ կողմերով որոշվող եռանկյունների մակերեսները՝ կիրառելով եռանկյան մակերեսը Հերոնի բանաձևի օգնությամբ հաշվող ֆունկցիա:
5. Տառային պարամետրերի ցանկացած քվային արժեքների համար հաշվել և արդյաժեղ արված արգահայտության արժեքը: Մեծագույն և փոքրագույն արժեքների հաշվումն իրագործել ֆունկցիայով:
 - ա) $y = \text{max}(a, a + b, a - b) + \text{max}(b, 2b - a, b + 2a)$,
 - բ) $y = \text{min}(3a, 2b, c) + \text{min}(a, b, 3c)$,
 - գ) $y = \text{max}(5, a, b) + \text{max}(7, b, a + b)$:

§ 1.17 ՉԱՆԳՎԱԾԸ ՈՐՊԵՍ ԵՆԹԱԾՐԱԳՐԻ ՊԱՐԱՄԵՏՐ

Եթե ենթածրագրում որպես ֆորմալ պարամետր որևէ չափի զանգված է կիրառվում, ապա անհրաժեշտ է ընդհանուր ծրագրի նկարագրությունների բաժնում տվյալ զանգվածի համար նոր տիպ նկարագրել:

Օրինակ՝

`TYPE vec=ARRAY[1..100] OF REAL;`

`matric=ARRAY[1..10,1..5] OF CHAR;`

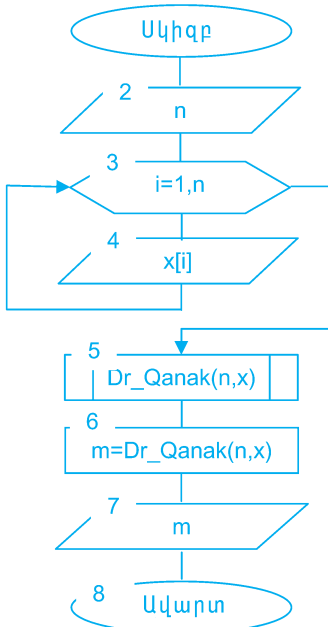
`PROCEDURE nor (x:vec;a:matric);`

և այլն:

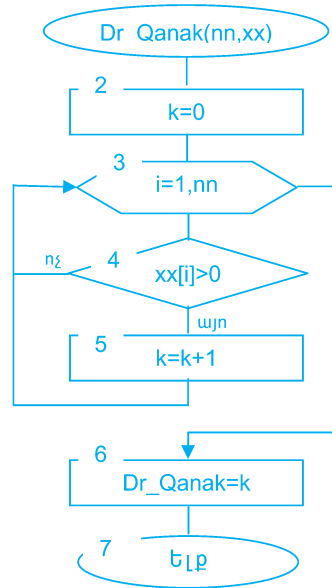
Ենթածրագրում մնան տիպի ֆորմալ պարամետր հայտարարելիս արդեն պետք է օգտվել `vec`, `matric` և այլ նոր տիպերից. բանն այն է, որ *Պասկալի* կոմպիլյատորը ֆորմալ պարամետր հայտարարելիս չի թույլատրում հիմնվել այլ տիպի վրա. չէ՞ որ այս դեպքում (օրինակ՝ `ARRAY[1..10] OF REAL`) նոր հայտարարվող զանգվածը ստեղծվում է՝ հիմնվելով `1..10` միջակայքային տիպի վրա:

Ասվածը պարզաբանելու նպատակով դիտարկենք հետևյալ խնդիրը. *Ֆունկցիայի միջոցով հաշվել ցանկացած n հատ ամբողջ տիպի տարրեր պարունակող միաչափ զանգվածի դրական տարրերի քանակը, որտեղ n-ը ստեղծաչափից ներմուծված [2;100] միջակայքի ցանկացած ամբողջ թիվ է:*

Նախ, ինչպես միշտ, կազմենք խնդրի լուծման բլոկ-սխեման.



ա) դեկլարող բլոկ-սխեմա



բ) ֆունկցիայի բլոկ-սխեմա

Նկ. 1.23. Միաչափ զանգվածը որպես ֆունկցիայի պարամետր

Ղեկավարող բլոկ-սխեմայում զանգվածի տարրերը ներմուծելուց հետո 5-րդ բլոկով կանչվել է *Dr_Qanak*(*nn,xx*) ֆունկցիան, որի վերադարձրած արժեքը 6-րդ բլոկում վերագրվել է *m*-ին: Ֆունկցիայի բլոկ-սխեմայում զանգվածի դրական տարրերի *k* քանակն արդեն հայտնի ալգորիթմով հաշվելուց հետո 6-րդ բլոկում վերագրվել է ֆունկցիայի անվանը՝ *Dr_Qanak*-ին:

Գրենք համապատասխան ծրագիրը.

```

PROGRAM Zang_Qanak;
TYPE vec=array[1..100] OF INTEGER;           {0_0}
VAR x:vec; i,n,m:BYTE;
      FUNCTION Dr_Qanak(nn:BYTE;xx:vec):BYTE; {0}
      VAR i,k:BYTE;
      BEGIN k:=0;
            FOR i:=1 TO nn DO
            IF xx[i]>0 THEN k:=k+1;
            Dr_qanak:=k                       {00}
            END;
BEGIN                                         {000}
  REPEAT WRITE('n= '); READ(n)              {1}
  UNTIL (n>1)AND(n<=100);                  {2}
  FOR i:=1 TO n DO READ(x[i]);
  m:=Dr_Qanak(n,x);                         {3}
  WRITELN('դրական տարրերի քանակը= ',m)
END.

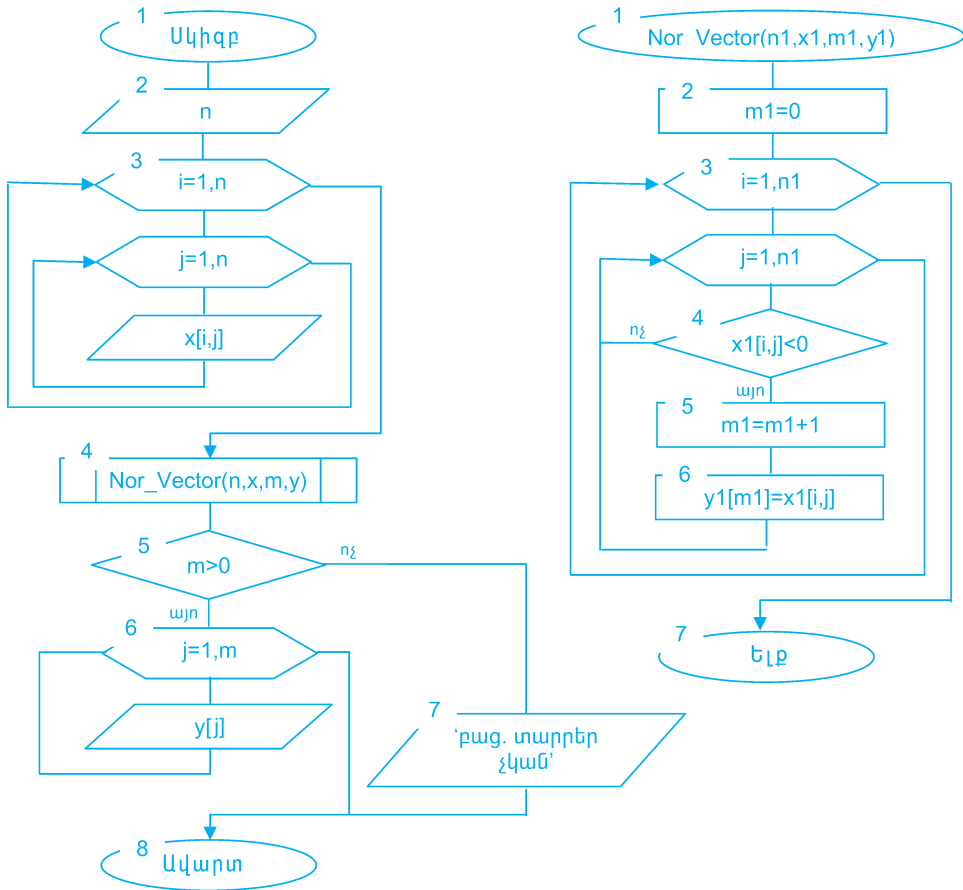
```

Ծրագրի {0_0} մակագրությամբ տողում հայտարարվել է *vec* տիպը, որը միաչափ զանգված է բնութագրում: Նպատակն այն է, որ ֆունկցիայի վերնագրում հնարավորություն ունենանք որպես ֆորմալ պարամետր միաչափ զանգված տալ: {0} տողում գրված է ֆունկցիայի վերնագիրը, որտեղ ֆունկցիայից վերադարձվող արժեքը նկարագրվել է *BYTE* տիպի: {00} տողում կատարվել է *Dr_Qanak:=k*; վերագրումը, որպեսզի պահանջված քանակը ֆունկցիան կարողանա վերադարձնել իրեն կանչող ծրագրին: {000} տողով սկսվել է ծրագրի հիմնական մասը: Այստեղ {1}-{2} տողերում ներառված *REPEAT-UNTIL* հետապայմանով ցիկլի օգնությամբ *n*-ը, որը զանգվածի տարրերի քանակն է, ներմուծվել է այնպես, որպեսզի [2,100] միջակայքի որևէ թիվ լինի (քանի որ {0_0} տողում հայտարարված զանգվածը կարող է ամենաշատը 100 տարր պարունակել): {3}-րդ տողում կանչվել է *Dr_Qanak* ֆունկցիան, որին փոխանցվել է թե՛ զանգվածը, թե՛ դրա տարրերի քանակը:

Հաջորդ խնդիրը լուծենք պրոցեդուրայի կիրառմամբ:

Տրված $n \times n$ (n -ը [2;10] միջակայքի ցանկացած թիվ է) իրական տիպի տարրեր պարունակող երկչափ զանգվածի բացասական տարրերից միաչափ զանգված ստանալ: Նոր զանգվածի ստացման գործընթացն իրականացնել պրոցեդուրայի միջոցով:

Կազմենք խնդրի լուծման բլոկ-սխեման.



ա) ղեկավարող բլոկ-սխեման

բ) պրոցեդուրայի բլոկ-սխեմա

Նկ. 1.24. Պրոցեդուրայի օգնությամբ երկչափից միաչափ զանգվածի սրացման ալգորիթ

Ղեկավարող բլոկ-սխեմայում զանգվածի տարրերը ներմուծելուց հետո 4-րդ բլոկով կանչվել է *Nor_Vector* պրոցեդուրան, որին ուղարկվել է $n \times n$ չափի երկչափ զանգվածը, իսկ արդյունքում ստացվել է m տարր պարունակող y միաչափ զանգվածը: Քանի որ ստացված զանգվածը կարող էր դատարկ լինել (եթե երկչափ զանգվածը բացասական տարրեր չպարունակեր), ապա պայմանի բլոկում (5) ստուգվել է $m > 0$ պայմանը՝ դրա ճշմարիտ արժեքի դեպքում միաչափ զանգվածը դատարկ չէ, և այն արտածվել է, հակառակ դեպքում արտածվել է համապատասխան հայտարարություն (7):

Պրոցեդուրայի բլոկ-սխեմայում ստացվելիք զանգվածի տարրերի նախնական քանակին 2-րդ բլոկով վերագրվել է 0, այնուհետև, դիտարկելով երկչափ զանգվածի յուրաքանչյուր տարր, առանձնացվել են բացասականներն ու 5 և 6-րդ բլոկների միջոցով վերագրվել $y1$ զանգվածի հերթական տարրին: Այսպիսով, պրոցեդուրայի ավարտին $m1$ տարր պարունակող $y1$ միաչափ զանգված կունենանք:

Կազմենք ծրագիրը.

```

PROGRAM Matric_Vektor;
TYPE matric=ARRAY[1..10,1..10]OF REAL;           {1}
    vector=ARRAY[1..100]OF REAL;
VAR i,j,n,m:BYTE; x:matric;y:vector;           {2}
PROCEDURE Nor_Vector(n1:BYTE;x1:matric;VAR m1:BYTE;VAR y1:vector); {3}
VAR i,j :BYTE ;
BEGIN m1 :=0 ;
    FOR i:=1 TO n1 DO FOR j:=1 TO n1 DO
    IF x1[i,j]<0 THEN
        BEGIN
            m1:=m1+1;
            y1[m1]:=x1[i,j]
        END
    END;
END;
BEGIN REPEAT READ(n) UNTIL (n>1)AND(n<=10);    {4}
FOR i:=1 TO n DO FOR j:=1 TO n DO READ(x[i,j]);
Nor_Vector(n,x,m,y);                             {5}
IF m>0 THEN FOR i:=1 TO m DO WRITELN('y[' ,i, ']= ' ,y[i]:4:1)
ELSE WRITELN('զանգվածում բացասական տարրեր չկան')
END.

```

Ծրագրի {1} մեկնաբանությամբ տողում *TYPE* առանցքային բառի տակ հաջորդաբար հայտարարվել են *matric* և *vector* նոր տիպերը, քանի որ պրոցեդուրայում անհրաժեշտ է այդ տիպերով ֆորմալ պարամետրեր նկարագրել: {2} տողում հայտարարվել են ծրագրի գլոբալ փոփոխականները (այն փոփոխականները, որոնք կարելի է կիրառել ծրագրի ցանկացած մասում): {3} տողում նկարագրվել է պրոցեդուրայի վերնագիրը, որտեղ *n1* և *x1* փոփոխականները հայտարարվել են որպես արժեք պարամետրեր (մուտքային), իսկ *m1* և *y1*-ը՝ փոփոխական պարամետրեր (ելքային):

Ծրագիրը սկսում է աշխատել {4} տողից, որտեղ ներմուծվում է զանգվածի չափը (*n*), ապա $n \times n$ տարր պարունակող զանգվածը: Ներմուծումից հետո {5} տողում կանչվել է *Nor_Vector* պրոցեդուրան: Եթե *y* զանգվածը դատարկ չէ՝ $m > 0$, ապա այն արտածվել է, հակառակ դեպքում հայտարարություն է արտածվել այն մասին, որ երկչափ զանգվածում բացասական տարրեր չկան:

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ **Ենթածրագրում զանգվածը որպես ֆորմալ պարամետր նկարագրելիս պետք է տալ նաև դրա մեջ մտնող տարրերի քանակը:**
- ◆ **Ենթածրագիրն առավել արագագործ կլինի, եթե զանգվածը որպես պարամետր նկարագրելիս այն որպես փոփոխական պարամետր նկարագրվի:**



1. Ի՞նչն է պարճառը, որ ենթածրագրում ֆորմալ պարամետր ներկայացնող զանգվածը նկարագրելիս պահանջվում է նախօրոք զանգված նկարագրող փիպ հայտարարել:
2. Կազմել հավելված 3-ի այս թեմային առնչվող խնդիրների լուծման բլոկ-սխեմաներն ու ծրագրերը:

§ 1.18 ՏՈՂԱՅԻՆ ՏԻՊԻ ՏՎՅԱԼՆԵՐԻ ՄՇԱԿՈՒՄ

Ծրագրավորման մեջ իր կիրառմամբ ուրույն տեղ ունի տողային տիպը, որը *Պասկալում* սահմանված կառուցվածքային տիպերից է: Տողային տիպը նախատեսված է տեքստային ինֆորմացիա մշակելու նպատակով:

Տողային տիպի փոփոխականն ըստ հայտարարման եղանակի կարող է տարբեր քանակությամբ պայմանանշաններ պարունակել: Եթե տողը տրվել է *s:string*; հայտարարմամբ, ապա այն կարող է մինչև 255 պայմանանշան պարունակել, որը տողային փոփոխականի առավելագույն չափն է: Եթե տողային փոփոխականը հայտարարվել է, օրինակ, *ss:string[50]*; եղանակով, ապա այս ձևով տրված տողը կարող է ամենաշատը 50 պայմանանշան պարունակել: Ընդ որում [] փակագծերում կարելի է 256-ից փոքր ցանկացած դրական ամբողջ թիվ տալ:

Տողային հաստատունը ներկայացվում է ապաթարցերի մեջ վերցված պայմանանշանների հաջորդականության տեսքով: Օրինակ՝ ‘Armen’, ‘4ag5h’ և այլն: Տողը կարող է լինել դատարկ՝ եթե այն ոչ մի պայմանանշան չի պարունակում. օրինակ՝ *s:=*; վերագրմամբ *s*-ը, որը մինչև 255 պայմանանշան կարող էր պարունակել՝ դատարկ է:

Ինչպես երևում է *ss*-ի վերը բերված հայտարարումից՝ տողը կարելի է ինչ-որ իմաստով *CHAR* տիպի միաչափ զանգվածի նմանեցնել՝

```
s:ARRAY[0..255] OF CHAR;
ss:ARRAY[0..k] OF CHAR;                                {1}
```

այն տարբերությամբ, որ տողային տիպի փոփոխականի երկարությունը կարող է տրված սահմաններում փոփոխվել, մինչդեռ միաչափ զանգվածինը, ինչպես գիտեք, հաստատուն է: Ցանկացած դեպքում տողային փոփոխականի բաղադրիչին կարելի է դիմել ինչպես զանգվածի տարրին, օրինակ, *WRITE(s[2])*; հրամանով կարտածվի *s* տողի երկրորդ պայմանանշանը:

Տողում 0 համարով պայմանանշանը հատուկ նշանակություն ունի. այն տվյալ պահին տողի մեջ առկա պայմանանշանների փաստացի քանակն է ցույց տալիս: Եթե, օրինակ, կատարվի *s:= 'abc'*; վերագրումը, ապա *WRITE(ORD(s[0])*; հրամանով կարտածվի 3 թիվը, որը *s* տողում առկա պայմանանշանների քանակն է: Եթե {1}

հայտարարությամբ տրված ss տողին փորձ արվի k -ից մեծ քանակությամբ պայմանաճանճան պարունակող տող վերագրել, ապա k -րդին հաջորդող պայմանաճանճաններն ուղղակի կանտեսվեն՝ առանց այդ մասին հաղորդագրության արտածման: Այսպիսով, օրինակ, եթե k -ն հավասար է 4, ապա $ss:= 'abcdef'$; վերագրմանը հաջորդող $WRITE(ss)$; հրամանը կարտածի $abcd$ ինֆորմացիան:

Տողային փոփոխականի արժեքը ստեղծաշարից ներմուծելու համար պետք է $READLN$ կիրառել (քանի որ տողային տիպի փոփոխականի երկարությունը կարող է ծրագրի կատարման ընթացքում փոփոխվել, և համակարգիչը պետք է տողի վերջը ճանաչելու հնարավորություն ունենա. $READLN$ -ը, ի տարբերություն $READ$ -ի, տողային փոփոխականի վերջին կցում է տողավերջի ' $\backslash 0$ ' հատուկ պայմանաճանճանը):

Պասկալում տողային տիպի մշակման նպատակով ստանդարտ ֆունկցիաներ կան: Ծանոթանանք դրանց:

$LENGTH(s)$ – ստանդարտ ֆունկցիա է, որը վերադարձնում է s տողում առկա պայմանաճանճանների քանակը: Օրինակ, եթե կատարվել է $VAR s:STRING[15]$; հայտարարումը, որին հաջորդել է $s:= 'xxx'$; վերագրումը, ապա $WRITE(LENGTH(s))$;-ի արդյունքում կարտածվի տողի փաստացի երկարությունը՝ 3, և ոչ թե 15, որն, ըստ հայտարարության, s -ի հնարավոր առավելագույն երկարությունն է:

$LENGTH$ -ի կիրառմամբ լուծենք հետևյալ խնդիրը.

Խ Ն Դ Ի ր 1. *Չափվել s տողում առկա a պայմանաճանճանների քանակը:*

```
PROGRAM Tox_1;
VAR s:STRING; n,i,l:BYTE;
BEGIN READLN(s);
      n:=LENGTH(s);           {1}
      l:=0;
      FOR i:=1 TO n DO IF s[i]='a' THEN l:=l+1;
      Writeln(l)
END.
```

$CONCAT(s1,s2, \dots, sn)$ – ստանդարտ ֆունկցիա է, որը հաջորդաբար, ըստ գրված հաջորդականության, իրար է կցում $s1, s2, \dots, sn$ տողերն ու արդյունքում ստանում նոր տող: $s1, s2, \dots, sn$ տողերի ընթացիկ երկարությունների գումարը պետք է փոքր կամ հավասար լինի 255-ից, իսկ եթե ստացված արդյունարար տողը պետք է վերագրվի մեկ այլ, օրինակ, ss տողի, ապա փոքր կամ հավասար պետք է լինի ss -ին ըստ հայտարարման տրված երկարությունից (հակառակ դեպքում՝ «ավելորդ» պայմանաճանճանները վերջից կանտեսվեն):

$CONCAT(s1, s2, \dots, sn); l s1+ s2+ \dots + sn$; գրառումները համարժեք գործողություններ են իրականացնում:

Օրինակ՝ եթե $s1:= 'a'$; $s2:= 'bc'$; , ապա $WRITE(CONCAT(s1,s2))$; և $WRITE(s1+s2)$; հրամաններով կարտածվի միևնույն abc ինֆորմացիան:

Խ Ն Դ Ի ր 2. *Ներմուծել անուն, հայրանուն և ազգանուն ներկայացնող $s1, s2$ և $s3$ տողերն ու ստանալ անվան, հայրանվան ու ազգանվան համակցությունը ներկայացնող նոր տող:*

```

PROGRAM Tox_2;
VAR      s1,s2,s3: STRING[15];
          s4:STRING[50];

BEGIN
WRITE('ներմուծեք անունը'); READLN(s1);      {պետք է ներմուծել մինչև 15
                                                պայմանանշան պարունակող անուն}
WRITE('ներմուծեք հայրանունը'); READLN(s2);  {պետք է ներմուծել մինչև 15
                                                պայմանանշան պարունակող հայրանուն}
WRITE('ներմուծեք ազգանունը'); READLN(s3);  {պետք է ներմուծել մինչև 15
                                                պայմանանշան պարունակող ազգանուն}
s4:=CONCAT(s1,'_','_',s2,'_','_',s3);
WRITE(s4)
END.

```

COPY(s,k,m) – ստանդարտ ֆունկցիան վերադարձնում է տող, որը ստացվում է s տողի k դիրքից սկսած m պայմանանշանի պատճենմամբ: Օրինակ, եթե $s := 'abbcc'$; ապա $WRITE(COPY(s,2,4))$; հրամանով կարտաձվի $abbc$ տողը:

Խնդիր 3. Տրված է 27 պայմանանշան պարունակող s տողը: s -ի 3-ին բազմապատիկ համար ունեցող պայմանանշանների կցումից նոր տող ստանալ:

```

PROGRAM Tox_3;
VAR s:STRING[27]; s1:STRING[9];
BEGIN
    REPEAT                                     {1}
        READLN(s)
    UNTIL LENGTH(s)=27;                       {2}
    s1:= "";                                  {3}
    FOR i:=1 TO 9 DO
        s1:=s1+COPY(s, 3 * i, 1);           {4}
    WRITELN(s1)
END.

```

Քանի որ, ըստ խնդրի պայմանի, տրված տողը պետք է 27 պայմանանիշ պարունակի, ապա ծրագրի {1}-{2} տողերում *REPEAT... UNTIL* ցիկլով պահանջված երկարությամբ տող է ներմուծվում: {3}-ում նոր ստանալիք $s1$ տողը նախապես դատարկվում է: Այժմ *FOR i:=1 TO 9 DO* ցիկլում *COPY* ֆունկցիայի միջոցով ստանալով s տողի 3-ին բազմապատիկ պայմանանշանների պատճենները, դրանք գումարման միջոցով կցվում են իրար՝ կազմելով պահանջվածը:

POS(s1,s2) ստանդարտ ֆունկցիան վերադարձնում է $s2$ տողի այն պայմանանիշի համարը, որտեղից սկսած $s2$ տողում $s1$ տողի պարունակությամբ ենթատող կա: Եթե $s2$ -ի մեջ $s1$ տողը չի պարունակվում, ապա *POS* ֆունկցիան վերադարձնում է 0 արժեք: Եթե $s1$ տողից $s2$ -ի մեջ մի քանի հատ կա, ապա *POS*-ի բազմակի կիրառումից արդյունքը չի փոխվի, միշտ կլինի նույնը՝ $s2$ -ում $s1$ -ի առաջին անգամ հանդիպելու համարը: Օրինակ, եթե

$s := 'abbsabbd';$
 1 2 3 4 5 6 7 8

ապա $WRITE(POS('bb',s));$ հրամանով կարտածվի 2 թիվը, ու հաջորդ $WRITE(POS('bb',s))$ -ի կրկնակի կիրառումից հետո էլ սինտույն 2 թիվը կարտածվի և ոչ թե հաջորդ 'bb'-ի համարը՝ 6:

Խ ն դ ի ր 4. Եթե տողում գոնե մեկ հատ *a* պայմանանշան կա, արտածել *YES*, հակառակ դեպքում՝ *NO* բառը:

```
PROGRAM Tox_4;
VAR s:STRING;
BEGIN  READLN(s);
      IF POS('a',s)>0 THEN WRITELN('YES') ELSE WRITELN('NO')
END.
```

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ Երկու տողերի հետք համեմատման =, <, >, <=, >=, <> գործողությունները կատարվում են առանձին պայմանանշաններով՝ չափից աջ հաջորդականությամբ: Եթե տողերից մեկը պարունակում է ավելի քիչ սիմվոլներ, քան մյուսը, ապա կարճ տողում թերի սիմվոլները փոխարինվում են $CHR(0)$ արժեքներով:



1. Ո՞րն է տողային փոփոխականի և սիմվոլային փոփոխականի զանգվածի նմանությունն ու տարբերությունը:
2. Տողային փոփոխականն ամենաշատը քանի՞ պայմանանշան կարող է պարունակել:
3. Ի՞նչ է պարունակում տողային փոփոխականի 0-ական տարրը:
4. Ի՞նչ է ցույց տալիս $LENGTH$ ֆունկցիան՝
 - ա) տողի հնարավոր առավելագույն երկարությունը,
 - բ) տողի ընթացիկ երկարությունը:
5. Կատարվել է $s:STRING[10];$ հայտարարությունը. ի՞նչ կարտածվի $WRITELN(s)$ -ի արդյունքում, եթե
 - ա) $s:=CONCAT('abc','kkk','mmm')$
 - բ) $s:='mmm'+'kkkk'+'ccc'$
6. Եթե $s:='abcdef';$, ապա ի՞նչ կվերադարձնի $COPY(s,3,2)$ ֆունկցիան՝
 - ա) 'de'
 - բ) 'cd'
7. Եթե $s1:='aaa';$ և $s2:='baada';$, ապա ի՞նչ է վերադարձնում $POS(s1,s2)$ ֆունկցիան՝
 - ա) 0,
 - բ) 2:
8. Եթե $x:='123';$ $y:='a23bcd';$ ապա ի՞նչ կստացվի $CONCAT(x,COPY(y,LENGTH(x),2))$ ֆունկցիայի արդյունքում:

§ 1.19 ՏՈՂԱՅԻՆ ՓՈՓՈԽԱԿԱՆՆԵՐԻ ՄՇԱԿՄԱՆ ՍՏԱՆԳԱՐՏ ՊՐՈՑԵԴՈՒՐԱՆԵՐ

Բացի ստանդարտ ֆունկցիաներից, *Պասկալի* գրադարանը տողային փոփոխականի հետ աշխատող նաև ստանդարտ պրոցեդուրաներ ունի:

Ուսումնասիրենք դրանց աշխատանքը:

DELETE(*s,k,m*) ստանդարտ պրոցեդուրան *s* տողի *k* դիրքից սկսած *m* քանակությամբ պայմանանշան է *հեռացնում* (ջնջում): Օրինակ՝ եթե *s:= 'abbbc'*; ապա **DELETE(*s,2,3*)**; հրամանից հետո **WRITE(*s*)**;-ը կարտածի *ac* ինֆորմացիան (*bbb-ն* ջնջվել է):

Այս պրոցեդուրայի աշխատանքին ծանոթանանք հետևյալ խնդրի միջոցով.

Խ ն դ ի թ 1. *Տրված s տողից հեռացնել դրանում առկա 'b' պայմանանշանները:*

```
PROGRAM Tox_1;
VAR s:STRING; k:BYTE;
BEGIN
    READLN(s);
    WHILE (POS('b',s)>0) DO
        BEGIN
            k:=POS('b',s);
            DELETE(s, k, 1)
        END;
    WRITELN(s)
END.
```

Օրինակ, եթե *s:= 'abdebbam'*; , ապա **WHILE**-ի առաջին կատարման ընթացքում կատարվի *k=2*, իսկ **DELETE**-ից հետո՝ *s='adebbam'* տողը (առաջին *b*-ն կջնջվի): **WHILE**-ի երկրորդ կրկնությունից հետո կստանանք *k=4* ու *s='adebam'*՝ արժեքները և վերջում՝ **WHILE**-ի երրորդ կրկնությունից հետո, *k=4* և *s='aseam'*՝ տողը, որն այլևս *b* պայմանանշան չի պարունակի և ծրագրի աշխատանքը կավարտվի:

INSERT(*s1,s2,k*) – ստանդարտ պրոցեդուրան *s2* տողի *k*-րդ դիրքից սկսած տեղադրում է *s1* տողը: Օրինակ, եթե *s1:= 'aa'*; *s2:= 'bbbb'*; , ապա **INSERT(*s1,s2,3*)**-ից հետո **WRITE(*s2*)**; հրամանով կարտածվի *bbaabb* ինֆորմացիան:

Եթե **INSERT**-ի արդյունքում արդյունարար տողի երկարությունը գերազանցում է ըստ հայտարարման դրան տրված առավելագույն երկարությունը, ապա վերջից «ավելացածները» հեռացվում են՝ առանց այդ մասին հայտնելու: Օրինակ, եթե տրված է *s2:STRING[5]*; հայտարարությունը և *s2:= 'aabbc'*; , ապա **INSERT('dd',s2,4)**; պրոցեդուրայի արդյունքում ստացված *s2*-ը հավասար կլինի *'aabdd'*՝ տողին, այսինքն՝ *s2*-ի վերջին *bc* պայմանանշանները դուրս կմնան:

Խ ն դ ի թ 2. *Տրված s տողում առկա a պայմանանշանները փոխարինել cc պայմանանշաններով:*


```

PROGRAM Tox_2;
VAR s:STRING[100]; k:BYTE;
BEGIN REPEAT
    READLN(s);
    UNTIL LENGTH(s)<= 50;
    WHILE POS('a',s)>0 DO
        BEGIN k:=POS('a', s);
            DELETE(s, k, 1);
            INSERT('cc', s, k)
        END;
    WRITELN(s)
END.

```

Քանի որ ծրագրում s տողը հայտարարվել է այնպես, որ այն կարող է ամենաշատը 100 պայմանանշան պարունակել, ապա ստեղծման գործընթացը կազմակերպվել է այնպես, որ ներմուծված տողը լինի մինչև 50 երկարության: Խնդիրն այն է, որ եթե ներմուծված տողը բաղկացած լինի միայն a պայմանանշաններից, ապա, ըստ խնդրի, պահանջվող նոր տողը կատացվի առավելագույն՝ 100 պայմանանշան պարունակող:

STR(x,s) ստանդարտ պրոցեդուրան ամբողջ կամ իրական տիպի x թիվը վեր է ածում դրան համարժեք տողային մեծության: Օրինակ, եթե $s:string$; է, ապա $STR(56,s)$; հրամանից հետո s -ի մեջ կհայտնվի '56' տողային մեծությունը: Եթե x -ն իրական թիվ է, ապա կարելի է տալ փոխակերպման ընթացքում պահանջվող ճշտության չափն այնպես, ինչպես արտածման $WRITE$ հրամանում, օրինակ, $STR(5.2:4:2,S)$; հրամանից հետո s -ում կհայտնվի '5.20' ինֆորմացիան (:4-ը ընդհանուր դիրքերի քանակն է, :2-ը՝ տասնորդական կետին հաջորդող պայմանանշանների քանակը):

Խնդիր 3. *Տրված է 10 իրական տարր պարունակող x միաչափ զանգվածը: Չանգվածի յուրաքանչյուր տարր փոխակերպել տողային համարժեքի՝ տասնորդական կետից հետո 2 նիշ ճշտություն պահպանելով:*

```

PROGRAM Tox_3;
VAR s:STRING; i:BYTE;
    x:ARRAY[1..10] OF REAL;
BEGIN FOR i:=1 TO 10 DO
    BEGIN READLN(x[i]);
        STR(x[i]:10:2,s);
        WRITELN('x[' ,i, ']= ',s)
    END
END.

```

Ի տարբերություն STR -ի՝ **VAL(s,x,c)** ստանդարտ պրոցեդուրան s տողային փոփոխականի արժեքը փոխակերպում է թվային արժեքի և պահպանում x -ում: Եթե s -ում առկա ինֆորմացիան իրոք x թվային փոփոխականի տիպի որևէ թվի տողային հա-

մարժեքն է, ապա փոխակերպումից հետո c -ն պարունակում է 0 թիվը, հակառակ դեպքում՝ 0 -ից տարբեր այլ ամբողջ թիվ: Օրինակ, եթե $s := '12'$; և x -ը *INTEGER* տիպի է, ապա $VAL(s,x,c)$ -ի արդյունքում x -ը կստանա 12 , իսկ c -ն՝ 0 արժեք, մինչդեռ, եթե $s := '12.1'$, ապա c -ն կստանա 3 արժեքը (այն դիրքի համարը, որտեղից առաջացել է սխալը), իսկ x -ի արժեքը կլինի անորոշ:

Խ ն դ ի ր 4. *Տրված է 10 տարր պարունակող տողային տիպի s միաչափ զանգված: Ստանալ տրամաբանական տիպի 10 տարր պարունակող b միաչափ զանգվածը, որի i -րդ տարրը TRUE է, եթե տրված զանգվածի i -րդ տարրը իրական թիվ է, հակառակ դեպքում՝ FALSE է:*

```
PROGRAM Tox_4;
VAR  s:ARRAY[1..10] OF STRING; c:BYTE; d:REAL;
      b:ARRAY[1..10] OF BOOLEAN;
BEGIN FOR i:=1 TO 10 DO
      BEGIN READLN(s[i]);
            VAL(s[i],d,c);
            IF c=0 THEN b[i]:=TRUE ELSE b[i]:=FALSE;
            WRITELN(b[i])
      END
END.
```

ՕՉՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ *VAL(s,k,c) պրոցեդուրային փոփոխիչ s փողի իմաստալից պայմանանշաններից առաջ եղած բացատրանիշներն անփոփոխում են, մինչդեռ s -ի վերջում փոփոխող բացատրանիշների առկայությունը կհանգեցնի սխալի:*
- ◆ *UPCASE(c) ֆունկցիան վերադարձնում է c սիմվոլային փոփոխականում պահպանված լատինական փոքրատառին համապատասխանող մեծատառը:*



1. *Եթե $x := '12345'$; , ապա ի՞նչ արժեք կընդունի x -ը DELETE(x,3,1); պրոցեդուրայի կատարման արդյունքում:*
2. *Եթե $x := '123'$, $y := 'abcd'$, ապա ի՞նչ արժեք կընդունի y -ը INSERT(x,y,3)-ի արդյունքում:*
3. *Կազմել հավելված 3-ի այս թեմային առնչվող խնդիրների լուծման բլոկ-սխեմաներն ու ծրագրերը:*

§ 1.20 ԳՐԱՌՈՒՄՆԵՐ

Մինչև հիմա *Պասկալ լեզվից* ուսումնասիրված կառուցվածքային տիպերն այն առանձնահատկությունն են ունեցել, որ միատիպ տարրեր են պարունակել: Մինչդեռ հաճախ խնդիրներ լուծելիս անհրաժեշտ է լինում մեկ ընդհանուրի մեջ տարբեր տիպերի տարրեր միավորել: *Պասկալ* լեզվում այդ նպատակով կիրառվում է **գրառում տիպը**:

Գրառումը կառուցվածքային տիպ է, որը կարող է կազմվել տարբեր տիպերի մեծությունների համակցումից:

Գրառում տիպը նկարագրվում է հետևյալ ընդհանուր եղանակով.

TYPE

իդենտիֆիկատոր = RECORD

բաղկացուցիչ դաշտերի նկարագրություններ

END;

որտեղ *իդենտիֆիկատորը* սահմանվող տիպի անվանումն է, *RECORD*-ն ու *END*-ը առանցքային բառեր են, իսկ *բաղկացուցիչ դաշտերի նկարագրություններն* իրարից կետ-ստորակետերով բաժանված իդենտիֆիկատորների հայտարարություններ են:

Օրինակ՝

TYPE ashakert = RECORD

anun: STRING[10];

azgamun: STRING[15];

matyani_hamar: BYTE

END;

VAR dasaran: ARRAY[1..30] OF ashakert;

x: ashakert;

Այստեղ սահմանված *ashakert* տիպը երեք բաղադրիչներ՝ *բաղկացուցիչ դաշտեր* ունի, որոնցից առաջին երկուսը տողային տիպի են՝ նախատեսված աշակերտի անվան ու ազգանվան համար, իսկ երրորդ դաշտը *BYTE* տիպի մեծություն է, որը բնորոշում է դասամատյանում աշակերտի ունեցած համարը:

Այնուհետև *VAR*-ի ներքո հայտարարված *dasaran*-ը *ashakert* տիպի 30 տարր պարունակող զանգված է, իսկ *x*-ը՝ *ashakert* տիպի առանձին փոփոխական:

Գրառում տիպի փոփոխականի *դաշտերին դիմելու երկու եղանակ կա*.

ա) *նշել փոփոխականի անունն ու բաղկացուցիչ դաշտի անվանումը՝ դրանք իրարից անջարեղով կետով (.), օրինակ, x.anun:= 'Armen'; dasaran[1].anun:= 'Anna';*

բ) *WITH* օպերատորի կիրառմամբ՝ հետևյալ կերպ՝

WITH գրառում տիպի փոփոխական DO օպերատոր,

որտեղ *WITH*-ն ու *DO*-ն առանցքային բառեր են, իսկ *DO*-ին հաջորդող *օպերատորը՝ Պասկալի* ցանկացած օպերատոր է կամ բլոկ: Օրինակ՝

WITH x DO anun:= 'Levon';

WITH dasaran[2] DO azgamun:= 'Arakelyan';

Գրառման դաշտերին դիմելու նշված եղանակները համարժեք են:

Գրառում տիպի փոփոխականին կարելի է նույն տիպի այլ փոփոխականի արժեք վերագրել, օրինակ, *dasaran[5]:=x*;

Որպես գրառում տիպի բաղադրիչ դաշտ կարող է հանդես գալ նաև նախապես հայտարարված մեկ այլ գրառում տիպի փոփոխական. այս դեպքում մնան դաշտ պարունակող գրառում տիպն անվանում են ներդրված դաշտերով գրառում:

Օրինակ՝

```
TYPE dproc= RECORD
```

```
    hamar:BYTE;
```

```
    y:ARRAY[1..100] OF ashakert
```

```
END;
```

```
VAR d:dproc;
```

Այստեղ *dproc* գրառում տիպը որպես բաղադրիչ պարունակում է նախօրոք տրված *ashakert* տիպը և այսպիսով ներդրված տիպ է հանդիսանում: Ընդ որում՝ ներդրված դաշտի բաղադրիչ տարրին կարելի է դիմել, օրինակ, հետևյալ կերպ՝ *d.y[2].anun:='Mari'*; այսինքն՝ կետի (.) միջոցով, կամ որ նույնն է՝ *WITH*-ի միջոցով.

```
WITH d DO y[2].anun:='Mari';
```

Գրառում տիպի հետ տարվող աշխատանքին ավելի մոտիկից ծանոթանալու նպատակով որևէ խնդիր լուծենք:

Խնդիր 1. *Տրված է n ($2 \leq n \leq 100$) տարր պարունակող զանգված, որի տարրերը հետևյալ բաղադրիչներով գրառումներ են՝*

ա) անունը,

բ) ազգանունը,

գ) ինֆորմատիկայից տարեկան միջր:

Անհրաժեշտ է որոշել աշակերտների ինֆորմատիկայից ունեցած գնահատականների միջին թվաբանականը:

```
PROGRAM Grarum_1;
```

```
TYPE mmm=RECORD
```

```
    an:string[10];
```

```
    azg:STRING[13];
```

```
    nish:BYTE
```

```
END;
```

```
VAR x:ARRAY[1..100] OF mmm;
```

```
i,n:BYTE; s:REAL;
```

```
BEGIN REPEAT
```

```
    READLN(n)
```

```
UNTIL (n > 1) AND (n <= 100);
```

```
FOR i:=1 TO n DO WITH x[i] DO
```

```
    BEGIN    READLN(an);
```

```
            READLN(azg);
```

```
            READLN(nish)
```

```
    END;
```

```

s:=0;
FOR i:=1 TO n DO s:=s+x[i].nish;
s:=s/n;
WRITELN('s= ',s:5:2)

```

END.

Խնդիր 2. Տրված է n ($2 \leq n \leq 100$) տարր պարունակող զանգված, որի տարրերը գրառումներ են հետևյալ բաղադրիչ դաշտերով՝

- ա) անունը,
- բ) ազգանունը,
- գ) ատեստատի միջինը,
- դ) ընդունելության քննությունը,

որտեղ ընդունելության քննությունն իր հերթին գրառում է հետևյալ դաշտերով՝

- ա) հայերենի նիշ,
- բ) մաթեմատիկայի նիշ,
- գ) պատմության նիշ:

Արտածել այն դիմորդների անուններն ու ազգանունները, ում ատեստատի միջինը բարձր է շեմային d արժեքից, քննական միջինը՝ տրված k -ից, իսկ մաթեմատիկայի նիշը՝ բարձր է 17-ից:

```
PROGRAM Grarum_2;
```

```
TYPE qnnakan= RECORD
```

```
    hayeren:BYTE;
```

```
    mathem:BYTE;
```

```
    patm:BYTE
```

```
END;
```

```
dimord= RECORD
```

```
    anun:STRING[10];
```

```
    azg:STRING[13];
```

```
    atestat:REAL;
```

```
    bal:qnnakan
```

```
END;
```

```
VAR x:ARRAY[1..100] OF dimord;
```

```
    i,n:BYTE; d,k:REAL;
```

```
BEGIN REPEAT READLN(n)
```

```
    UNTIL (n>1)AND(n<=100);
```

```
    READLN(d,k);
```

```
    FOR i:=1 TO n DO WITH x[i] DO
```

```
        BEGIN READLN(anun);
```

```
            READLN(azg);
```

```
            READLN(atestat);
```

```
            READLN(bal.hayeren);
```

```
            READLN(bal.mathem);
```

```
            READLN(bal.patm)
```

```
        END;
```

```
FOR i:=1 TO n DO WITH x[i] DO
  IF (atestat>d)AND((bal.hayeren+ bal.mathem+ bal.patm)/3>k)AND
    (bal.mathem>17) THEN WRITELN(anun, '  ' ,azg)
```

END.

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ Պատկալում գրառման տիպի նկարագրության մեջ թույլատրվում է տարբերակային դաշտ կիրառել: Այդ նպատակով օգտվում են CASE ... OF առանցքային բառերից, որոնք, սակայն, ընտրության օպերատոր չկազմելով՝ END-ով չեն ավարտվում: Գրառման մեջ տարբերակային դաշտը պետք է լինի միակը և հաջորդի գրառման մնացած դաշտերի նկարագրություններին:



1. Ի՞նչ է գրառումը:
2. Ինչպե՞ս են նկարագրում գրառումը:
3. Ի՞նչ նպատակով են կիրառում WITH օպերատորը:
4. Կազմել հավելված 3-ի այս թեմային առնչվող խնդիրների լուծման բլոկ-սխեմաներն ու ծրագրերը:

§ 1.21 ՖԱՅԼԵՐ

Ֆայլը համակարգչի արտաքին հիշող սարքի վրա անուն ունեցող տիպային է, կամ էլ՝ ինֆորմացիա ներմուծելու կամ արտածելու նպատակին ծառայող տրամաբանական սարք:

Մենք ուսումնասիրելու ենք ֆայլը որպես արտաքին հիշող սարքի վրա ստեղծվող և պահպանվող միատիպ բաղադրիչների հավաքածու, որն ունի անուն և որոշակի ծավալ, որը սահմանափակվում է միայն կրիչի ազատ հիշողությամբ: Ֆայլի բաղադրիչները կարող են Պատկալում սահմանված ցանկացած տիպի լինել, բացի ֆայլայինից:

Ֆայլը կարելի է հայտարարել հետևյալ երեք եղանակներով՝

ա) TEXT;

բ) FILE:

գ) FILE OF տիպ:

Ֆայլի հայտարարման եղանակը բնորոշում է դրանում պահպանվելիք տարրերի

տիպը: Եթե ֆայլը հայտարարվել է որպես *TEXT*, ապա այն նախատեսված է տեքստային ինֆորմացիայի պահպանման համար: **Տեքստային ֆայլի** բաղադրիչները տարբեր երկարությամբ տողեր են: Տեքստային ֆայլը հաջորդական մշակման ֆայլ է՝ ֆայլի *k*-րդ բաղադրիչը հասանելի է դառնում միայն նախորդ՝ *k-1*-րդ տարրը մշակելուց հետո:

Եթե ֆայլը հայտարարվել է որպես *FILE*, ապա կոչվում է **չտիպայնացված**: Չտիպայնացված ֆայլի առանձնահատկությունն այն է, որ կարող է տարբեր տիպերի տվյալներ պարունակել:

Մենք առավել մոտիկից կձանոթանանք ու կաշխատենք, այսպես կոչված, **տիպայնացված** ֆայլերի հետ:

Տիպայնացված ֆայլը հայտարարվում է հետևյալ ընդհանուր եղանակով՝

VAR իդենտիֆիկատոր:FILE OF տիպ;

որտեղ *իդենտիֆիկատորը*, այսպես կոչված, **ֆայլային փոփոխական** է, որը *Պասկալի* ծրագիրն արտաքին ֆայլի հետ կապող միջոց է: *FILE*-ը և *OF*-ը առանցքային բառեր են, իսկ *տիպը Պասկալում* սահմանված ցանկացած տիպ է, բացի ֆայլայինից:

Պասկալի միջավայրից ֆայլ ստեղծելու համար կիրառվում է հետևյալ ստանդարտ պրոցեդուրան՝

ASSIGN(ֆայլային փոփոխական, տողային փոփոխական կամ հաստատում);

որտեղ *ֆայլային փոփոխականը* վերը նկարագրված եղանակներից որևէ մեկով հայտարարված փոփոխական է: *Տողային փոփոխականը* կամ *հաստատումը* ֆայլի անվանումն է: *Ֆայլի անվանումը* համակարգչի օպերացիոն համակարգի կողմից ընդունված կանոններով կառուցված տող է՝ կարող է ներառել լատինական այբուբենի մեծատառերն ու փոքրատառերը, @ \$ % ^ & | # () ~ - _ ‘ պայմանանշաններն ու թվանշանները: Ֆայլի անունը կարող է պարունակել նաև ընդլայնում՝ մինչև երեք պայմանանշան ներկայացնող հաջորդականություն, որը ֆայլի անունից բաժանվում է կետով (.): Ֆայլի անունը կարող է սկսվել արտաքին կրիչի անունից (*A, B, C, D, E* և այլն), որին հաջորդում է երկու կետը (:): Ֆայլի լրիվ անվանումը կարող է ներառել նաև այն թղթապանակի հասցեն, որտեղ ստեղծվում է:

Օրինակ, ֆայլի ճիշտ անվանումներ են՝

‘ab.txt’

‘C:\TP\BIN\My_file.dat’

‘A:\k.pas’

և այլն:

ASSIGN պրոցեդուրայի աշխատանքի արդյունքում, եթե նշված հասցեում տվյալ անվամբ ֆայլ չկա, ապա *սրեղծվում է*, որից հետո ֆայլային փոփոխականն ամրագրվում է ստեղծված ֆայլին: Այսուհետև այդ ֆայլի հետ ցանկացած գործողություն իրականացնելու նպատակով ֆայլի անվան փոխարեն կիրառվելու է ֆայլային փոփոխականը: Եթե պարզվում է, որ *ASSIGN*-ում բերված անվամբ ֆայլ արդեն գոյություն ունի, ապա ուղղակի ֆայլային փոփոխականն ամրագրվում է դրան:

ASSIGN պրոցեդուրայից հետո ֆայլի հետ հետագա աշխատանք տանելու համար պետք է այդ «ֆայլը բացել»: Ֆայլը «բացել» նշանակում է տալ դրա հետ աշխատելու նպատակը՝ *իվյալների գրանցում* կամ *ընթերցում*:

Եթե ֆայլը նոր է ստեղծվում, ապա, բնականաբար, այն բացվում է տվյալներ գրանցելու նպատակով: Այս դեպքում ֆայլը բացվում է **REWRITE** պրոցեդուրայով, որի ընդհանուր գրելաձևը հետևյալն է՝

REWRITE(ֆայլային փոփոխական);

որտեղ *ֆայլային փոփոխականը* տվյալ ֆայլին *ASSIGN*-ով ամրագրված փոփոխականն է: Եթե *REWRITE*-ը կիրառվում է արդեն գոյություն ունեցող ֆայլի համար, ապա տվյալ *ֆայլի պարունակությունը ոչնչանում է* առանց որևէ հաղորդագրության արտածման, որից հետո ֆայլը նախապատրաստվում է նոր տվյալներ գրանցելու:

Ֆայլից առկա *իվյալներն* ընթերցելու նպատակով *ֆայլը բացում են RESET* պրոցեդուրայով, որն ունի հետևյալ ընդհանուր տեսքը՝

RESET(ֆայլային փոփոխական);

որտեղ *ֆայլային փոփոխականը* *ASSIGN*-ի միջոցով տվյալ ֆայլին ամրագրված փոփոխականն է: Այս պրոցեդուրայի արդյունքում տվյալ ֆայլի հետ աշխատող հատուկ ցուցիչը, այսպես կոչված՝ *ֆայլի մարկերը*, բերվում է ֆայլի սկիզբ և ցույց է տալիս ֆայլի առաջին՝ 0-ական համարով տարրի վրա: Եթե *RESET*-ով փորձ արվի բացել գոյություն չունեցող (ջնջված) ֆայլ, ապա ներկառուցված *IORESULT* (մուտքի/ելքի գործողության արդյունք) ֆունկցիան 0-ից տարբեր արժեք կվերադարձնի: *RESET* պրոցեդուրայով թույլատրվում է բացված ֆայլից ոչ միայն տվյալներ ընթերցել, այլև, անհրաժեշտության դեպքում, տվյալներ գրանցել: Այսպիսով, հնարավորություն է ստեղծվում ֆայլը խմբագրելու, այնտեղ նոր տվյալներ ավելացնելու:

Տիպայնացված ֆայլում տվյալները գրանցվում են *WRITE* պրոցեդուրայի միջոցով, որի ընդհանուր գրելաձևը հետևյալն է՝

WRITE(ֆայլային փոփոխական, գրանցվող փյալներ);

Որպես *գրանցվող փյալներ* կարող են լինել մեկ կամ իրարից ստորակետերով անջատված մի քանի փոփոխականներ, որոնց տիպը պետք է համընկնի ֆայլային փոփոխականը հայտարարելիս ֆայլի տարրերին տրված տիպի հետ: Օրինակ, եթե տրված են

f:FILE OF INTEGER;

a,b:INTEGER;

հայտարարություններն ու

a:=2;

b:=SQR(a)+5;

վերագրումները, ապա *WRITE(f,a,b)*; պրոցեդուրայով մարկերի ընթացիկ դիրքում ֆայլի մեջ նախ կգրանցվի 2 արժեքը, որից հետո մարկերը կտեղաշարժվի հաջորդ դիրքի վրա, որտեղ էլ կգրանցվի 9 արժեքը (2^2+5): Այսպիսով, յուրաքանչյուր տարր գրանցելուց հետո ֆայլային մարկերն ավտոմատ անցում է կատարում հաջորդ դիրքին:

Ֆայլում գրանցված տվյալներն ընթերցելու նպատակով կիրառվում է **READ** պրոցեդուրան, որի ընդհանուր գրելաձևը հետևյալն է.

READ(ֆայլային փոփոխական, ընթերցվող տվյալներ);

որտեղ *ընթերցվող տվյալները* ֆայլային փոփոխականի տիպի մեկ կամ միմյանցից ստորակետերով անջատված մի քանի փոփոխականներ են: Օրինակ՝ *READ(f,a,b)*; պրոցեդուրայի արդյունքում ֆայլային մարկերի ընթացիկ դիրքում առկա տվյալն ընթերցվելով կվերագրվի *a*-ին, ապա մարկերն ավտոմատ անցում կատարելով հաջորդ տվյալի վրա՝ այն կընթերցի ու կգրանցի *b*-ում:

Տիպայնացված ֆայլերի հետ աշխատելիս հնարավոր է *ֆայլային մարկերը տեղադրել* ֆայլի ցանկացած տվյալի վրա: Դրա համար կիրառվում է **SEEK** պրոցեդուրան, որի ընդհանուր գրելաձևն այսպիսին է՝

SEEK(ֆայլային փոփոխական, քաղաղորդիչ համար);

որտեղ *քաղաղորդիչ համարը* ֆայլի անհրաժեշտ տարրի համարն է (ֆայլի առաջին տարրի համարը 0-ն է):

Ֆայլի մեջ առկա տարրերի քանակը որոշելու նպատակով կիրառվում է **FILESIZE** ֆունկցիան հետևյալ ընդհանուր գրառմամբ՝

FILESIZE(ֆայլային փոփոխական);

Այսպես, օրինակ, որպեսզի ֆայլի մարկերը տեղավորենք առկա ֆայլի վերջում՝ առաջին ազատ դիրքի վրա, կարելի է գրել

SEEK(f,FILESIZE(f));

որտեղ *f*-ը ֆայլային փոփոխականն է:

Ֆայլի որևէ տարրին հաջորդող տարրերը ջնջելու համար կիրառում են հետևյալ պրոցեդուրան՝

TRUNCATE(ֆայլային փոփոխական);

Երբեմն անհրաժեշտ է լինում իմանալ, թե ֆայլային մարկերը ֆայլի n° քաղաղորդիչի վրա է: Այդ նպատակով օգտվում են **FILEPOS** ֆունկցիայից, որի տեսքը հետևյալն է՝

FILEPOS(ֆայլային փոփոխական);

Այս ֆունկցիան վերադարձնում է **LONGINT** տիպի ամբողջ թիվ:

Ֆայլի հետ աշխատելուց հետո այն «փակում» են հատուկ պրոցեդուրայով, որը հետևյալն է՝

CLOSE(ֆայլային փոփոխական);

Այս պրոցեդուրան ապահովում է ստեղծվող ֆայլի տվյալների ամբողջականությունը: **CLOSE** պրոցեդուրայից հետո կարելի է շարունակել ֆայլի հետ աշխատել, քանի որ ծրագրի և ֆայլի կապը չի խզվում:

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ **READ** սրանդարտ պրոցեդուրայում առաջին պարամետրը (մուտքի ֆայլը) չնշելու դեպքում փվյալները ներմուծվում են սրեղնաշարից:
- ◆ **WRITE** սրանդարտ պրոցեդուրայում առաջին պարամետրը (ելքի ֆայլը) չնշելու դեպքում փվյալներն արտածվում են (գրանցվում են) էկրանին:
- ◆ **ASSIGN** պրոցեդուրայում ֆայլի անվանումը հասցեի հետ միասին կարող է մինչև 79 պայմանանշան պարունակել:



1. **Ի՞նչ է ֆայլը:**
2. **Պատկարում կիրառվող ֆայլի ի՞նչ տիպեր գիտեք. ինչպե՞ս են դրանք հայտարարվում:**
3. **Տիպայնացված ֆայլերի հետ կիրառվող ի՞նչ սրանդարտ ենթածրագրեր գիտեք:**

§ 1.22 ՖԱՅԼԵՐԻ ԱՇԽԱՏԱՆՔԸ ՍՊԱՍԱՐԿՈՂ ՕԺԱՆԴԱԿ ԵՆԹԱԾՐԱԳՐԵՐ

Երբեմն անհրաժեշտ է լինում *Պատկալի* ծրագրից ոչ միայն ֆայլ, այլև թղթապանակ ստեղծել: Դրան ծառայող ստանդարտ պրոցեդուրան ունի հետևյալ տեսքը.

MKDIR(թղթապանակ):

որտեղ *թղթապանակը* ստեղծվող թղթապանակի հասցեն և անվանումը պարունակող տողային արտահայտություն է, օրինակ, 'C:\tp\nor'. այստեղ *nor*-ը ստեղծվող նոր թղթապանակի անվանումն է (ենթադրվում է, որ *tp*-ում *nor* անունը կրող թղթապանակ մինչ այդ չկար):

Ավելորդ թղթապանակը ջնջելու համար նախատեսված է **RMDIR** պրոցեդուրան հետևյալ գրելաձևով`

RMDIR(թղթապանակ):

այստեղ *թղթապանակը* հեռացման ենթակա թղթապանակի հասցեն ու անվանումը ներկայացնող տողային արտահայտություն է: Հեռացվող թղթապանակը պետք է լինի դատարկ` ոչ մի ֆայլ կամ թղթապանակ չպարունակի: Այսինքն` թղթապանակը ոչնչացնելիս նախապես այն պետք է «դատարկել»` ջնջել դրանում առկա թղթապանակներն ու ֆայլերը:

Ֆայլ ջնջելու համար կիրառվող պրոցեդուրան ունի հետևյալ գրելաձևը`

ERASE(ֆայլային փոփոխական):

որտեղ *ֆայլային փոփոխականը* ջնջման ենթակա ֆայլի հետ **ASSIGN**-ով ամրագրված փոփոխականն է: Եթե ջնջվող ֆայլը մինչ այդ բացված է եղել **RESET** կամ

REWRITE պրոցեդուրաներից որևէ մեկով, ապա այն ջնջելուց առաջ անհրաժեշտ է փակել՝ կիրառելով *CLOSE* պրոցեդուրան:

Որոշ դեպքերում նպատակահարմար է լինում *Պասկալ* ծրագրից փոխել ֆայլի նախկին անվանումը, այլ խոսքով՝ *անվանափոխել ֆայլը*: Դրա համար կիրառում են

RENAME(ֆայլային փոփոխական, նոր անվանում);

ընդհանուր գրելաձև ունեցող ստանդարտ պրոցեդուրան. այստեղ *ֆայլային փոփոխականը* անվանափոխման ենթակա ֆայլին ամրագրված փոփոխականն է, իսկ *նոր անվանումը*՝ ֆայլի նոր անվանումը ներկայացնող տողային արտահայտություն: Այս պրոցեդուրան իրագործելուց առաջ ևս անհրաժեշտ է *CLOSE*-ի միջոցով նախապես փակել ֆայլը, եթե մինչ այդ այն բացվել էր *RESET*-ի կամ *REWRITE*-ի միջոցով:

Չանազան խնդիրներին ուղղված ծրագրերի դեպքում երբեմն անհրաժեշտություն է ծագում փոխել ընթացիկ (աշխատանքային) թղթապանակը, որի համար նախատեսված է

CHDIR(թղթապանակի ուղի);

ընդհանուր գրելաձևով պրոցեդուրան, որտեղ *թղթապանակի ուղին* նոր ընթացիկ թղթապանակի հասցեն է:

Ֆայլերի հետ կապված աշխատանքում հաճախ օգտակար ծառայություն է մատուցում *EOF* ֆունկցիան, որի ընդհանուր գրելաձևը հետևյալն է՝

EOF(ֆայլային փոփոխական);

Այս ֆունկցիայից վերադարձվող արժեքը տրամաբանական (*BOOLEAN*) տիպի է. հավասար է *TRUE*, եթե ֆայլային մարկերը ֆայլի վերջին տարրին հաջորդող ազատ դիրքի վրա է և *FALSE*՝ հակառակ դեպքում: Այսպիսով, եթե այս ֆունկցիան *TRUE* վերադարձնի ֆայլից տվյալներ ընթերցելիս, կնշանակի՝ ֆայլն ավարտվել է, իսկ ֆայլում տվյալներ գրանցելու գործընթացում՝ որ նոր տվյալը կգրանցվի ֆայլի վերջից:

Ֆայլերի աշխատանքին ծանոթանալու նպատակով լուծենք հետևյալ խնդիրը.

Խնդիր 1. *C սկավառակի եր թղթապանակում նախ n ամբողջ տիպի տարրեր պարունակող d1.dat անվամբ ֆայլ ստեղծել, ապա նույն թղթապանակում մեկ այլ d2.dat ֆայլ ստեղծել հետևյալ կերպ՝ d1.dat-ի յուրաքանչյուր տարրից հետո գրելով 0 թիվը:*

PROGRAM File_1;

VAR f1,f2:FILE OF INTEGER; {f1 և f2 ֆայլային փոփոխականների հայտարարում}
i,n,d,k:INTEGER;

BEGIN READ(n);

ASSIGN(f1,'C:\tp\d1.dat');

REWRITE(f1); {f1 ֆայլի սրեղծում և նախապատրաստում այնրեղ տարրեր գրանցելու}

FOR i:=1 TO n DO {f1 ֆայլում սրեղնաշարից ներմուծված n հար տարրերի գրանցում}

BEGIN READ(d);

WRITE(f1,d)

END;

CLOSE(f1);

```

ASSIGN(f2, 'C:\tp\d2.dat');
REWRITE(f2); {f2 ֆայլի սրեղծում և նախապատրաստումը փարթեր գրանցելու}
RESET(f1); {f1 ֆայլի բացում՝ ընթերցման նպատակով}
k:=0;
WHILE NOT EOF(f1) DO    {քանի դեռ f1 ֆայլի ավարտին չենք հասել՝
                        ընթերցել դրա յուրաքանչյուր փարթը և գրանցել f2-ի մեջ}
    BEGIN  READ(f1,d);
           WRITE(f2,d,k)
    END;
CLOSE(f2);                {1}
RESET(f2); {f2 ֆայլի ընթերցման նախապատրաստում}
WHILE NOT EOF(f2) DO    {քանի դեռ f2 ֆայլի ավարտին չենք հասել՝
                        ընթերցել յուրաքանչյուր փարթը և արտածել}
    BEGIN READ(f2,d);
           WRITELN(d)
    END;
CLOSE(f2);
CLOSE(f1)
END.

```

Եթե խնդիր դրված լիներ *d1.dat* ֆայլի մեջ ստանալ պահանջված պատասխանը, ապա կարելի էր խնդիրը բերված եղանակով լուծելուց {1} տողից հետո ավելացնել հետևյալ ծրագրային կտորը՝

```

CLOSE(f1)
ERASE(f1);
RENAME(f2, 'd1.dat');

```

Խնդիր 2. *C* սկավառակի *tp* թղթապանակում ստեղծված են *d1.dat* և *d2.dat* *n*-ական իրական տարրեր պարունակող ֆայլերը: *tp* թղթապանակում ստեղծել նոր՝ *d3.dat* ֆայլ, որի տարրերը ստացվում են առաջին երկու ֆայլերի միևնույն հերթական համարն ունեցող տարրերի գումարումից:

```

PROGRAM File_2;
VAR f1,f2,f3:FILE OF REAL;
    i,n:BYTE; a,b:REAL;
BEGIN ASSIGN(f1, 'C:\tp\d1.dat'); REWRITE(f1);
      ASSIGN(f2, 'C:\tp\d2.dat'); REWRITE(f2);
      READ(n);
      FOR i:=1 TO n DO {f1 և f2 ֆայլերի սրեղծում}
          BEGIN READ(a);
                 WRITE(f1,a);
                 READ(b);
                 WRITE(f2,b)
          END;

```

```

CLOSE(f1);
CLOSE(f2);
ASSIGN(f3), 'C:\tp\d3.dat');
REWRITE(f3); {f3 ֆայլի սրեղծում և նախապատրաստում այնպեղ տարրեր գրանցելու}
  RESET(f1);
  RESET(f2);
  FOR i:=1 TO n DO {f3 ֆայլում անհրաժեղ տվյալների սրացում}
    BEGIN
      READ(f1,a);
      READ(f2,b);
      WRITE(f3,a+b)
    END;
CLOSE(f3);
FOR i:=1 TO n DO {պատասխան հանղիսացղ f3 ֆայլի տարրերի սրտածում}
  BEGIN
    READ(f3,a);
    WRITELN(a)
  END
END.

```

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ **DISKFREE** (սկավառակ) ֆունկցիան վերաղարչնում է նշված սկավառակի վրա առղա ազատ հիղողղության ծավալը՝ բայթերղղ: Այստեղ սկավառակը նղելղու համար պեղթ է տաղ դրա թվային համարը, որտեղ *A* սկավառակին համապատասխանում է 1, *B*-ին՝ 2, *C*-ին՝ 3 և այղն թվերը:
- ◆ **DISKSIZE** (սկավառակ) սրանղարտ ֆունկցիան վերաղարչնում է նշված սկավառակի ծավալը՝ բայթերղղ: Այստեղ նղղնպես սկավառակը տրղղում է վերը նշված թվային համարներղղ:



1. Պասկաղղում ֆայղերի աղղատանըը սպասարղղղ ի՞նղ օժանղակ ենթածրաղղեր գղղեր:
2. Կաղղնղ հավեղղված 3-ի այս թեղնային աղղնղղղղ խնղղրների լղղծման բղղկ-սխեղնաներն ող ծրաղղերը:

2.

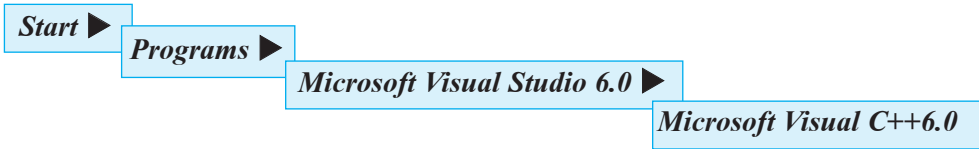
ԾՐԱԳՐԱՎՈՐՄԱՆ C++ ԼԵԶՎԻ ՀԻՄՈՒՆՔՆԵՐԸ



§ 2.1 C++ ԾՐԱԳՐԻ ԱՇԽԱՏԱՆՔԱՅԻՆ ՄԻՋԱՎԱՅՐԸ

Ծրագրավորման լեզուները հիմնականում *ինկոդացված աշխատանքային միջավայր* են ունենում, որտեղ ծրագրային տեքստերը ստեղծվում, կարգաբերվում և անհրաժեշտության դեպքում իրագործվում են:

Visual C++-ի աշխատանքային միջավայր մտնելու համար անհրաժեշտ է հաջորդաբար կատարել հետևյալ քայլերը.



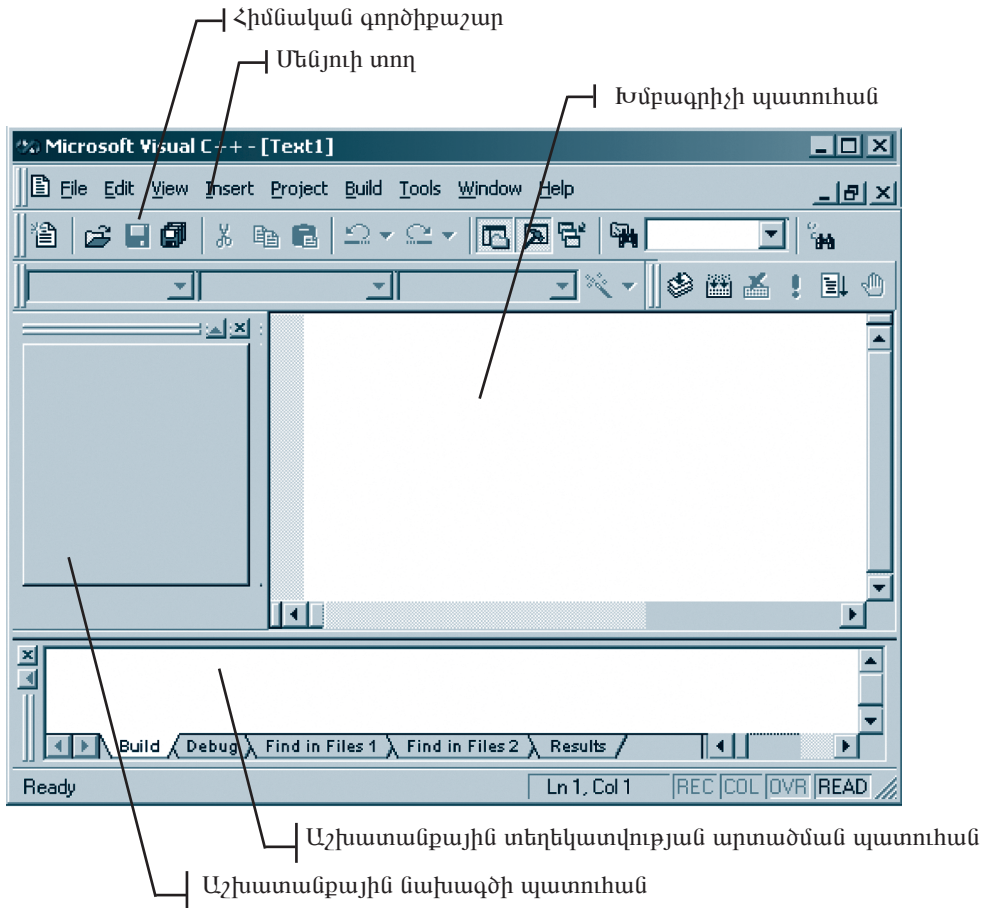
Արդյունքում էկրանին կբերվի C++-ի գլխավոր պատուհանը՝ աշխատանքային միջավայրը (նկ. 2.1):

Ինչպես նկատում եք, գլխավոր պատուհանը 3 բաղկացուցիչ մաս է ներառում.

- աշխատանքային նախագծի պատուհան (*Project workspace*), որն օգնում է բազմաթիվ ֆայլերից բաղկացած ծրագիր մշակելիս,
- խմբագրիչի պատուհան (*Editor*), որտեղ ներմուծվում և խմբագրվում է ծրագրի տեքստը,
- աշխատանքային տեղեկատվության արտածման պատուհան, ուր ծրագրի կոմպիլյացիայի (թարգմանման), կապակցման ու կատարման փուլերին առնչվող տեղեկատվություններ են արտածվում:

Գլխավոր պատուհանի մենյուի տողն ու հիմնական գործիքաշարը բազմաթիվ հրամաններ և գործիքներ են ներառում, որոնց կծանոթանանք ըստ անհրաժեշտության: Նշենք միայն, որ գործիքաշարի յուրաքանչյուր բաղադրիչի համար ենթատեքստային օգնություն կա. եթե մկնիկի ցուցիչը մոտեցնեք՝ դրան առնչվող տեղեկատվություն կստանաք:

Չնայած *Visual C++*-ը թույլատրում է աշխատել պատուհանային համակարգին հատուկ գրաֆիկական բարձրակարգ ձևավորմամբ միջավայրում, այդուհանդերձ, այստեղ նախատեսված, այսպես կոչված, *կոնսոլային ներդիրը* բեռնավորելիս օպերացիոն համակարգը *կոնսոլային պատուհան* է ձևավորում, որը արտաքնապես նման է *MS DOS* կամ այլ օպերացիոն համակարգերում հրամանային տողի ռեժիմով



Նկ. 2.1. Visual C++-ի գլխավոր պատուհան

աշխատելիս բացվող պատուհանին: Մենք կաշխատենք կոնսոլային ներդիրով, քանի որ այս ռեժիմն է առավել հարմար կիրառել C++ լեզուն ուսումնասիրելու համար:

Առաջարկում ենք սկսել C++-ին ծանոթանալ հետևյալ պարզագույն ծրագրով.

```
#include <iostream> //1
using namespace std; //2
void main() //3
{ //4
    cout<< "MY FIRST PROGRAM!!!" //5
} //6
```

Այստեղ բերված ծրագրի աջ մասում յուրաքանչյուր տողից հետո `//`-ով սկսվող գրառումը **մեկնաբանություն** է, որը մատչելի է դարձնում ծրագրի բացատրման գործընթացը: Առանց բացատրանիշի իրար հաջորդող երկու գծերի (`//`) միջոցով տրվում են ծրագրի տվյալ **պողիկ վերաբերող մեկնաբանությունները**, իսկ եթե մեկնաբանությունը մի քանի տող է զբաղեցնում, ապա այն սկսում են `/*` և ավարտում `*/` պայմանանշանների համակցությամբ:

//1 տողը ներառում է հատուկ հրահանգ, այսպես կոչված, *պրեպրոցեսորին* (*նախապրոցեսոր*) ուղղված *դիրեկտիվ*, որը ներմուծման և արտածման գործընթացն ապահովող որոշ լրացուցիչ ծրագրային միջոցներ է կցում գրված ծրագրին:

Պրեպրոցեսորային դիրեկտիվները ղեկավարում են ծրագրի տեքստի ձևափոխման գործընթացը՝ նախքան ծրագիրը քարզմանն էլ:

Ընդհանրապես *#include* հրահանգի միջոցով ծրագրի նախնական տեքստին արտաքին այլ, նախապես կազմավորված ֆայլեր են կցվում: Հաճախ դրանք *.h* ընդլայնում ունեցող ֆայլեր են, որոնք ընդգրկված են *include* թղթապանակում. այս ֆայլերն անվանում են *վերնագրային ֆայլեր*:

Վերնագրային ֆայլերը ASCII ձևաչափով սրեղծված գրադարանային ստանդարտ ֆայլեր են, որոնց պարունակությունը կարելի է դիտել էկրանին, իսկ ցանկության դեպքում՝ նույնիսկ տպել:

#include-ի միջոցով կարելի է նաև գրված ծրագրին ոչ ստանդարտ՝ ծրագրավորողի կողմից մշակված ֆայլ կցել. ֆայլի անվանումն այս դեպքում ներառվում է չափերաների “ “ մեջ: Օրինակ՝

```
#include "nor_file.cpp" կամ #include "banali.h" և այլն:
```

//2 – տողում կիրառված *std*-ն C++-ի ստանդարտ գրադարանում առկա *դարձ-ժողովրդական անվանումն* է: Ծրագրում *using namespace std;* գրառմամբ հասանելի են դառնում *std*-ում սահմանված անվանումները (մասնավորապես այստեղ է սահմանվել ծրագրում կիրառված *cout* -ը):

//3 – C++-ի ցանկացած ծրագիր պետք է իրագործման ելակետ հանդիսացող մաս ունենա, որն ունի *main()* անվանումը՝ դա ծրագրի, այսպես կոչված, *գլխավոր մասն* է:

Սովորաբար ծրագրավորողները մասնատում են ծավալուն ծրագրերն ըստ իրենց նշանակության վիճքի, համեմատաբար ինքնուրույն, C++-ում *ֆունկցիաներ* կոչվող մասերի (առավել մանրամասնորեն սրանց կանդադարանք հետագայում): Ընդհանրապես C++ լեզվում ցանկացած պարզագույն ծրագիր է ձևակերպվում որպես ֆունկցիա: *void*-ը *main()* գրառմամբ մեջ կիրառվել է նշելու համար, որ ծրագրի գլխավոր *main* անունը կրող ֆունկցիան արժեք չի վերադարձնում C++ ծրագիրը բեռնավորող օպերացիոն համակարգին:

//4 և // 6 տողերում առկա ձևավոր *{ }* փակագծերն օգտագործվում են ծրագրի բաղկացուցիչ հրամանները (օպերատորները) խմբավորելու համար. այստեղ խմբավորվել է *main*-ի մեջ ներառված միակ հրամանը:

Ձևավոր փակագծերում ներառված հրամանների համախումբն անվանում են *բլոկ կամ բաղադրյալ օպերատոր*:

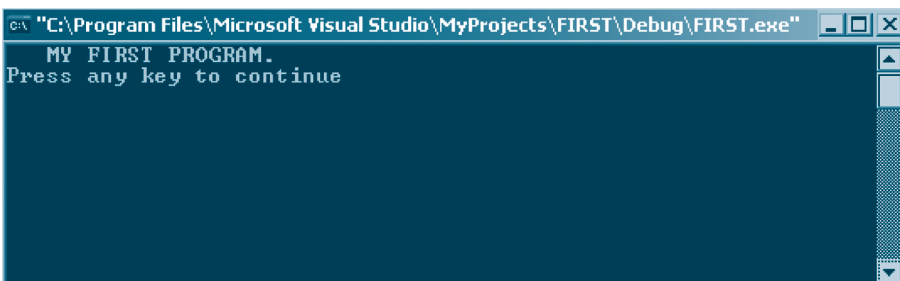
//5 տողով ըստ տրված `cout<<` հրահանգի համակարգիչը էկրանին կարտածի “*BAREV BOLORIN*” հաղորդագրությունը: Այստեղ `cout`-ը **էլքի սրանդարդ հոսքի** այն օբյեկտն է, որն անհրաժեշտ հաղորդագրությունն ուղղում է օպերացիոն համակարգի կողմից ընդունված ստանդարտ ելքային սարքին (էկրանին): `cout`-ին կից կրկնակի կիրառված փոքրի (<) նշանն ունի այն իմաստը, որ դրան հաջորդող ինֆորմացիան տեղադրվում է ելքային հոսքի (արտածման ենթակա տվյալների) մեջ. <<-ն անվանում են **փեղադրման գործողություն**: Ի տարբերություն դրա, `cin >>` հրահանգի միջոցով էլ իրագործվում է տվյալների ներմուծումը ստեղնաշարից: Այստեղ `cin`-ը **մուտքային սրանդարդ հոսքի** այն **օբյեկտն** է, որը ներմուծվող ինֆորմացիան ընդունում է մուտքի ստանդարտ սարքից՝ ստեղնաշարից, իսկ >>-ն այն իմաստն ունի, որ դրան հաջորդող տվյալները մտցվում են մուտքի հոսքի (ներմուծման ենթակա տվյալների) մեջ:

C++ լեզվով գրված ծրագիրն իրականացնելու համար ծրագրի նախնական տեքստը պետք է թարգմանել մեքենայական կոդի. այդ նպատակով կարելի է օգտվել գլխավոր պատուհանի **Build** մենյուի հրամաններից՝

- **compile** – թարգմանել խմբագրիչի պատուհանում առկա ակտիվ ծրագիրը,
- **build** – աշխատանքային նախագծի կապակցում (կոմպանովկա). թարգմանվող ծրագրին կցվում են նաև գրադարանային անհրաժեշտ ստանդարտ ծրագրերի տեքստերը:

Compile և **build** հրամանների իրագործման արդյունքում առաջացած հնարավոր սխալների մասին համակարգիչը հայտնում է **աշխատանքային փեղեկավորման արդյունքի Output** պատուհանում:

Թարգմանված և սխալներ չպարունակող ծրագիրը կարելի է իրագործել **Execute** հրամանով: Նկ. 2.2-ում տեսնում եք վերը բերված ծրագրի իրագործման արդյունքը.

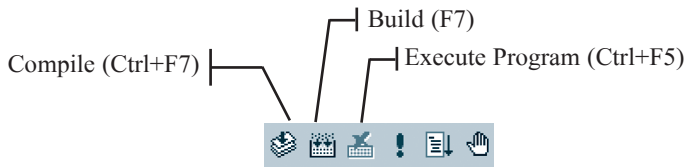


```

C:\Program Files\Microsoft Visual Studio\MyProjects\FIRST\Debug\FIRST.exe
MY FIRST PROGRAM.
Press any key to continue
  
```

Նկ. 2.2. Ծրագրի կադրման արդյունք

Գլխավոր պատուհանի (նկ. 2.1) հիմնական գործիքաշարի մեջ ծրագրի թարգմանության թվարկած միջոցներին համարժեք (նկ. 2.3) գործիքներ կան:



Նկ. 2.3. Build գործիքների վահանակ

C++ լեզվի նախորդ տարբերակների թարգմանիչները վերը բերված ծրագիրը չեն կարող թարգմանել. այդ դեպքում //1 և //2 տողերում ներառվածն անհրաժեշտ է փոխարինել տվյալ թարգմանիչներին «հասկանալի» `#include <iostream.h>` գրառմամբ: Հետագայում նման բարդություններից խուսափելու նպատակով կօգտվենք այս երկրորդ (`#include <iostream.h>`) տարբերակից:

Աշխատանքային նախագծի հետ աշխատանքն ավարտելու համար կարելի է.

- մենյուի տողի *File* ենթամենյուից ընտրել *Close Workspace* հրամանը,
- կոճակով փակել *Visual C++*-ի ներդիր պատուհանը:

ՕԳՏԱԿԱՐ Է ԻՄԱՆՆԱԼ

- ◆ **Պրեպրոցեսորային դիրեկտիվը պետք է սկսվի # պայմանանշանով, մեկ տող գրադեցնի և ավարտվի տողավերջի պայմանանշանով:**
- ◆ **C++ ծրագիրը որևէ նախագիծ սրեղծելու արդյունքում, կախված նախագծի կազմից՝ գեներացնում է Debug կամ Release թղթապանակներից որևէ մեկը և հետևյալ ընդլայնումներով ֆայլերը.**
 - ◆ `.dsw`, ◆ `.dsp`,
 - ◆ `.opt`, ◆ `.ncb`:
- ◆ **Սրեղծված ծրագրային նախագծում որևէ նախօրոք կազմավորված ֆայլ ավելացնելու համար անհրաժեշտ է.**
 - ◆ պարճենել ֆայլը նախագծի աշխատանքային թղթապանակի մեջ,
 - ◆ աշխատանքային նախագծի պատուհանում մկնիկի աջ սեղմակով ընկրել *Source Files* թղթապանակը,
 - ◆ բացված ենթատեսքային մենյուի մեջ ընկրել ֆայլ ավելացնելու համար նախատեսված *Add files to Folder* հրամանը,
 - ◆ բացված *Insert Files...* երկխոսային պատուհանում մկնիկով ընկրել ավելացման ենթակա ֆայլն ու ընկրությունը հաստատել *OK* կոճակով:
- ◆ **Նախկինում սրեղծված աշխատանքային նախագիծն ակտիվացնելու (բացելու) համար կարելի է.**
 - ◆ մտնել C++ միջավայր,
 - ◆ *File* ենթամենյուի մեջ ընկրել *Open Workspace* հրամանը,
 - ◆ բացված երկխոսային պատուհանում գտնել պահպանված նախագծի թղթապանակն ու այնտեղ փնտրել նախագծի անունը կրող `.dsw` ընդլայնումով ֆայլը,
 - ◆ մկնիկի աջ սեղմակով բացել այն

կամ՝

- ◆ մտնել C++ միջավայր,
- ◆ File ենթամենյուի մեջ ընտրել Recent Workspace հրամանը,
- ◆ եթե այնպեղ բերված ֆայլերի ցանկում ներառված է անհրաժեշտ .dsw ֆայլը՝ ընտրել այն

կամ՝

- ◆ առանց C++-ի միջավայր մտնելու գտնել անհրաժեշտ ֆայլն ու մկնիկով այն սկսել:



1. Ինչպե՞ս սկսել C++-ի աշխատանքային միջավայրը:
2. Ինչի՞ համար է օգտագործվում աշխատանքային տեղեկատվության արվածման պատուհանը:
3. C++ ծրագրում ինչպե՞ս են տրոհվում վերաբերող մեկնաբանություն ստեղծում:
4. Ի՞նչ է կապարվում #include <iostream.h> հրահանգով:
5. Ի՞նչ է կապարվում using namespace std հրահանգով:
6. Կարո՞ղ է C++-ով գրված ծրագիրը main() անունը կրող ծրագրային մոդուլ չպարունակել:
7. Ինֆորմացիան էկրանին արտածելու համար ի՞նչ միջոց է կիրառվում C++ լեզվում:
8. C++ միջավայրում ստեղծված ծրագիրն իրագործելու համար ի՞նչ հաջորդական փուլերի պետք է ենթարկվի:
9. C++ ծրագրի աշխատանքն ավարտելու քանի՞ եղանակ գիտեք:



Հաբորարոր աշխատանք

2.1

C++ ֆայլի ստեղծում

Աշխատանքի ընթացքում Alfa.cpp անվանմամբ ֆայլ ենք ստեղծելու, որի իրագործման արդյունքում էկրանին կարգավի աշակերպի անունը:

Հաջորդաբար իրականացրեք հետևյալ քայլերը.

1. Start մեկնարկային մենյուից հաջորդաբար ընտրելով Programs, Microsoft Visual Studio 6.0, Microsoft C++6.0 գրառումները՝ Visual C++-ի աշխատանքային միջավայր մտնել:
2. Էկրանին բերված պատուհանը փակել Close կոճակով:
3. C++ գլխավոր պատուհանում (նկ. 2.1) ընտրել File ենթամենյուի New հրամանն ու բացված մենյուից ընտրել Files ենթամենյուն:
4. Բացված ցուցակից ընտրել ֆայլի C++ Source File տիպը:
5. File Name դաշտում ներմուծել ֆայլի Alfa.cpp անունը:

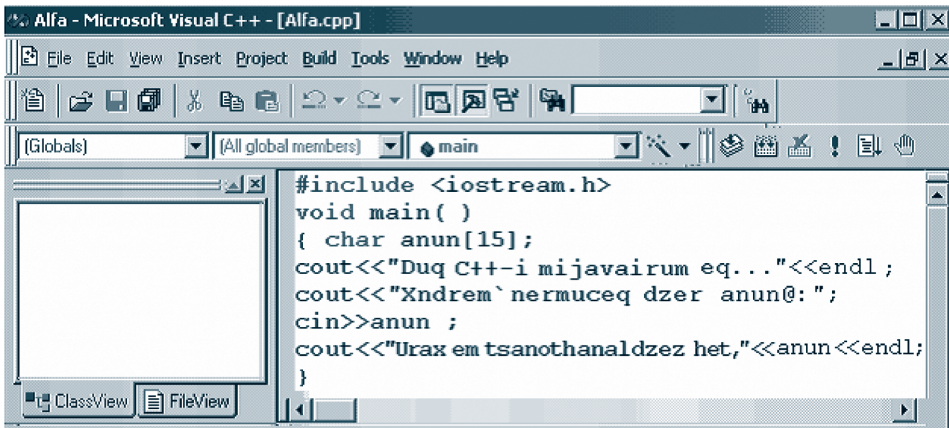
6. Կատարված ընդրությունն ավարտեք OK կոճակով:

Այժմ նախագծի խմբագրիչի պատուհանը պատրաստ է ծրագրի նախնական կողմն ընդունելու. համոզվեք, որ դրա վերին չախ անկյունում հայտնվեց բարձր պեքսային կուրսորի պատկերը:

7. Ներմուծեք հեղեղայ պեքսարը.

```
#include <iostream.h>
void main()
{ char anun[15];
cout << "Duq C++-i mijavairum eq..." << endl ;
cout << "Xndrem`nermuceq dzer anun@";
cin >> anun ;
cout << "Urax em tsanothanal dzez het, " << anun << endl;
}
```

Արդյունքում C++-ի գլխավոր պատուհանը կունենա հեղեղայ պեքսը.



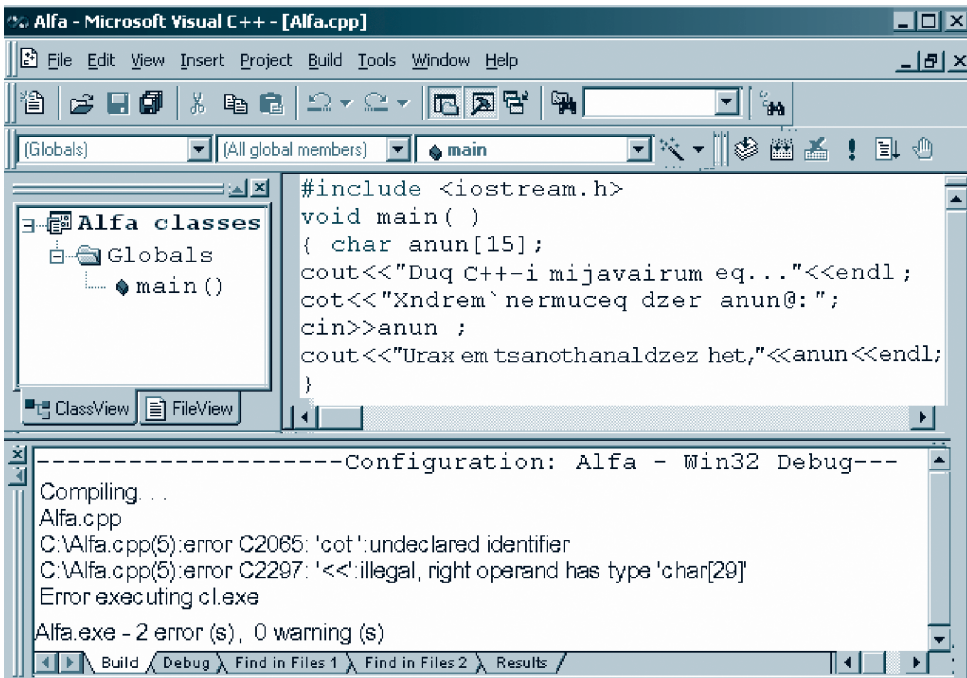
Այժմ բարգմանեք ներմուծված ծրագիրն ու իրագործեք այն.

8. Գործիքների վահանակից ընդրեք Compile (C) գործիքը կամ միաժամանակ սեղմեք սրեղնաչառի Ctrl և F7 սրեղները:

Եթե ծրագրի ներմուծման արդյունքում սխալ չեք թույլ տվել, ապա աշխատանքային տեղեկատվության պատուհանում կարգաձևի

Alfa.exe - 0 error(s), 0 warning(s)

հայտարարությունը, հակառակ դեպքում՝ սխալների ցանկը: Եթե որևէ սխալ է հայտնաբերվել՝ ծրագիրը չի կարող իրագործվել՝ նախքան դրանք ուղղելը: Օրինակ, եթե այս ծրագրի հերթական 5-րդ տողում cout-ի փոխարեն ներմուծվեց cot (սխալ, կոմպիլյատորի կողմից չճանաչված բառ), ապա Compile-ից հետո կունենալիք հեղեղայ պեքսերը.



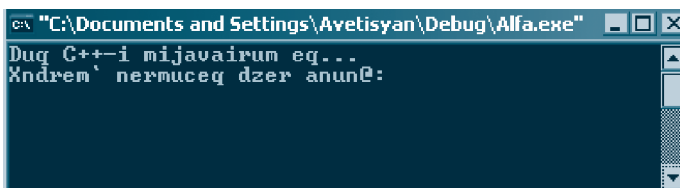
Միայն ուղղելու նպատակով մկնիկի շախ սեղմակի կրկնակի սեղմումով պեղք է ընկրել *Output* պատուհանում առկա սխալներից առաջինը. արդյունքում ներմուծված ծրագրում կընկրվի այն առաջին փողը, որի վրա կապարվել է փվյալ սխալը: Միայն ուղղելուց հետո կրկին պեղք է կիրառել *Compile-ը*: Այս ճանապարհով պեղք է ուղղել առկա հնարավոր մնացած սխալները, մինչև որ ներմուծվածը ճշգրտեմ համընկնի լաբորատոր աշխատանքի 7-րդ կեղում բերված փեքսի հեք: Այս քայլերը (հերթական սխալն ուղղելն ու *Compile* գործիքն ընկրելը) պեղք է կրկնել այնքան, որ *Output-ում* արպածվի 0 error(s), 0 warning(s) հայտարարությունը:

10. Այժմ գործիքների վահանակից ընկրեք *Build* գործիքը կամ սեղմեք սրեղնաշարի F7 ֆունկցիոնալ կոճակը. համոզվեք, որ աշխատանքային փեղեկափոխյան արպածման պատուհանում փրվեց

0 error(s), 0 warning(s)

հայտարարությունը:

11. *Գործիքների վահանակից* այժմ ընկրեք *Execute Program* գործիքը կամ միաժամանակ սեղմեք սրեղնաշարի *Ctrl* և *F5* սրեղները. եքե ամեն ինչ ճիշտ եք կատարել՝ էկրանին կբացվի կոնսոլային պատուհան, որը կունենա հետևյալ փեքը.



12. Սրբեղնաշարից ներմուծեք ձեր անունը և սեղմեք *Enter* սրբեղնը:
13. Արդյունքում կունենաք, օրինակ, հետևյալ պատուհանը

```

c:\ "C:\Documents and Settings\Avetisyan\Debug\Alfa.exe"
Dug C++-i mijavairum eq...
Xndrem' nermuceq dzer anu@Armen
Urax em tsanothanal dzez het, Armen
Press any key to continue
  
```

14. Այժմ կոնսոլային պատուհանը փակեք սեղմակով և վերադարձեք C++-ի գլխավոր պատուհան:
15. Ընկերեք մենյուի փողի *File* ենթամենյուի *Close Workspace* հրամանը. արդյունքում C++-ի գլխավոր պատուհանը կդադարակվի:
16. Նորից մտեք *File* ենթամենյու և ընկերեք *Recent Workspace* հրամանը. բերված ցուցակից ընկերեք *Alfa* նախագիծը՝ դրա վրա մկնիկի շախ սեղմակի կրկնակի սեղմում կատարելով:
17. Աշխատանքային նախագծի պատուհանի ներքևի մասում ընկերեք *File View* կոճակը. այժմ այն կստանա հետևյալ տեսքը՝



18. Մկնիկի ցուցիչով ընկերեք *Alfa files* անվանման շախ մասում եղած կոճակը. արդյունքում կրեսներ՝



19. *Alfa.cpp* անվան վրա մկնիկի շախ սեղմակի կրկնակի սեղմում կատարեք և ախախտով կրկին խմբագրիչի պատուհանում ակտիվացրեք ներմուծած ծրագիրը:
20. Աշխատանքն ավարտեք *Visual C++*-ի գլխավոր պատուհանի փակման կոճակով:

§ 2.2 C++ ԼԵԶՎԻ ՇԱՐՎՆԻՑՈՒՄԻԹՅՈՒՆԸ: ՈՒՆԱՐ ԳՈՐԾՈՂՈՒԹՅՈՒՆՆԵՐ

Ցանկացած լեզվով ծրագիր գրելիս օգտվում են լեզվի հիմնարար բաղադրիչ տարրերից: Ծանոթանանք C++ ծրագրավորման լեզվում կիրառվող հիմնական հասկացություններին ու օժանդակ տարրերին:

Լեզվում կիրառվող այբուբենը 96 պայմանանշան է պարունակում, որոնցից միայն 91-ն ունեն իրենց գրելաձևը: Գրելաձև չունեցող պայմանանշանները հետևյալն են.

- բացատանիշը,
- հորիզոնական տաբուլյացիան (հորիզոնական ուղղությամբ որոշակի քանակությամբ բացատանիշերի համախումբը),
- ուղղաձիգ տաբուլյացիան,
- նոր տողի սկիզբը (*Enter* ստեղծի գործողությանը համարժեք պայմանանշան),
- նոր էջի սկիզբը:

Գրելաձև ունեցող պայմանանշանները կազմում են.

- լատինական այբուբենի մեծատառերն ու փոքրատառերը,
- 0, 1, 2, ..., 8, 9 թվանշանները,
- հետևյալ 29 հատուկ նշանները. “ { } , | : / \ () + - / % \ ; ‘ & ^ ~ . * ? _ ! # < = > ”

Գրելաձև ունեցող պայմանանշանների միջոցով կազմվում են լեզվի, այսպես կոչված, **լեկսեմները**:

Լեկսեմը ծրագրային տեքստի միավոր է, որը ենթակա չէ տրոհման:

Լեկսեմները կարելի է խմբավորել հետևյալ կերպ.

- իդենտիֆիկատորներ,
- առանցքային (ծառայողական) բառեր,
- հաստատումներ (լիտերալներ),
- գործողությունների նշաններ,
- բաժանիչներ:

Իդենտիֆիկատորները նախատեսված են ծրագրում օգտագործվող մեծություններին անվանում տալու համար: C++ լեզվում իդենտիֆիկատորները կազմվում են լեզվում կիրառվող այբուբենի տառերով և թվանշաններով, իսկ գրելաձև ունեցող պայմանանշաններից կարող են պարունակել միայն տողատակի ընդգծման _ նշանը: Իդենտիֆիկատորները պարտադրաբար պետք է սկսվեն տառով, կարող են լինել ցանկացած երկարության, չնայած համակարգչի համար տարբերիչ են հանդիսանում առաջին 63 պայմանանշանները:

Ճիշտ կազմված իդենտիֆիկատորներ են, օրինակ, *abc*, *x1*, *a*, *d_56*, *AnunAzganun* իդենտիֆիկատորները:

C++ լեզվում սահմանված բառեր կան, որոնք հատուկ իմաստ ունեն և չեն կարող

կիրառվել այլ կերպ, քան նախասահմանված են: Այդ բառերն անվանում են **առանց-քային**: Առանցքային են, օրինակ, հետևյալ՝ *do, double, int, char, const* բառերը: Առանցքային բառերի ամբողջական ցանկը բերված է ձեռնարկի վերջում առկա Հավելված 1-ում:

Չնայած թույլատրվում է, սակայն խորհուրդ չի տրվում իդենտիֆիկատորները սկսել ընդգծման (`_`) նշանով կամ դրանցում երկու իրար հաջորդող ընդգծման նշաններ կիրառել, որովհետև նման եղանակով կազմված իդենտիֆիկատորները լեզվում այլ կիրառական նշանակություն ունեն:

Հաստատում լիպերայինները լեկսեմներ են, որոնք հաստատուն արժեքներ են ներկայացնում: Սրանք բաժանվում են հետևյալ խմբերի.

- ամբողջ,
- իրական,
- տրամաբանական,
- սիմվոլային,
- տողային:

Ամբողջ հաստատունը կարող է ներկայացվել **դասական, ութական** կամ **դասնվեցական** տեսքերով:

Տասական հաստատունը 0-ից 9-ը թվանշաններով կազմված հաջորդականություն է, որը չի սկսվում 0-ով՝ բացառությամբ 0 թվից: Օրինակ՝ 5, 17, 100, 0 և այլն: Ընդ որում՝ բացասական ամբողջ հաստատունները կազմվում են առանց նշանի ամբողջից՝ մինուսի (-) կիրառմամբ. օրինակ՝ -57, -200 և այլն:

Ութական ամբողջ հաստատունները կազմվում են 0-ից 7 թվանշաններով և սկսվում են 0-ով: Օրինակ՝ 063, 043, 043 և այլն:

Տասնվեցական հաստատունները կազմվում են 0, 1, ..., 9, A, B, C, D, E, F տասնվեցական նիշերով և սկսվում են 0x-ով:

Ամբողջ հաստատուն մեծությունները համակարգչի հիշողության մեջ տեղ գրավելով և ունենալով կոնկրետ արժեքներ՝ անուններ չունեն: Կախված հաստատունի թվային մեծությունից, համակարգիչը դրան վերագրում է C++ լեզվում ամբողջ թվերի համար սահմանված հետևյալ տիպերից մեկը.

Աղյուսակ 2.1

Բայթերի քանակը	Տիպը	Մեծության հնարավոր սահմանը
1	<i>char</i>	0-ից 255
2	<i>short</i>	-32768-ից 32767
2	<i>unsigned short</i>	0-ից 65535
2	<i>int</i>	-32768-ից 32767
2	<i>unsigned int</i>	0-ից 65535
4	<i>long</i>	-2147483648-ից 2147483647
4	<i>unsigned long</i>	0-ից 4294967295

Իրական հաստատումները կարող են կազմվել հետևյալ բաղադրիչներով.

- ամբողջ մաս (տասհիմնային ամբողջ հաստատում),
- ամբողջ և կոտորակային մասերն իրարից բաժանող տասնորդական կետ,
- կոտորակային մաս (տասհիմնային ամբողջ հաստատում),
- e կամ E պայմանանշան,
- տասական աստիճանի ցուցիչ (նշանով կամ առանց նշանի տասհիմնային ամբողջ հաստատում),
- F կամ f , L կամ l պայմանանշաններ:

Իրական թվերի գրառման մեջ կարող է բացակայել տասնորդական կետի աջ կամ ձախ մասերից ցանկացածը, բայց ոչ երկու մասերը միաժամանակ: Օրինակ, իրական թվերի ճիշտ գրառումներ են՝

.56, 5., 2E+6, 2.71 :

Կախված իրական հաստատումի արժեքից, համակարգիչը $C++$ -ով աշխատելիս այն դասում է **float**, **double** կամ **long double** տիպերից որևէ մեկի տիպի (աղյուսակ 2.2):

Աղյուսակ 2.2

Չափը (բայթերով)	Տիպը	Հնարավոր արժեքների տիրույթը
4	float	$3.4E-38$ -ից $3.4E+38$
8	double	$1.7E-308$ -ից $1.7E+308$
10	long double	$3.4E-4932$ -ից $1.1E+4932$

Տրամաբանական հաստատումը *true* կամ *false* արժեքներ ընդունող *bool* տիպի հաստատում է, որտեղ *false*-ին համապատասխանում է 0 թիվը, իսկ տրամաբանական արտահայտության արժեքը հաշվելիս 0-ից տարբեր մեծությունն ընդունվում է որպես *true*:

Միմվոլային հաստատումը ապաքարցերի մեջ վերցված *char* տիպի ցանկացած առանձին պայմանանշան է: Օրինակ՝ ‘a’, ‘+’, ‘#’ և այլն: Միմվոլային հաստատումին տրամադրվում է 1 բայթ ծավալով հիշողություն:

$C++$ լեզվում սահմանված են հատուկ իմաստ ունեցող որոշ պայմանանշաններ, որոնք սկսվում են հակադարձ թեք (\) գծով:

Աղյուսակ 2.3

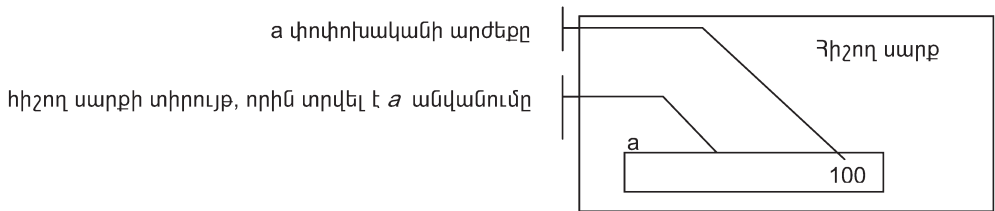
Նշանը	Գործողությունը
\\a	չայնային ազդանշան
\\b	մեկ նիշ երբ փանել կուրսորը
\\f	անցում նոր էջի
\\n	անցում նոր փողի սկիզբ
\\r	փողի սկիզբ
\\t	հորիզոնական փաթույլացիա
\\v	ուղղաձիգ փաթույլացիա
\\	հակադարձ թեք գիծ
\\'	ապաքարց
\\''	չակերտ
\\?	հարցական նշան

Տողային հասարակությունը չակերտների միջև առնված պայմանանշանների հաջորդականությունն է: Օրինակ՝ “*sa tox e ...*” : Տողային հաստատուն ստեղծելով՝ համակարգիչը դրա վերջում ավելացնում է ‘\0’ պայմանանշանը: Այսպիսով, “\0”-ն ներկայացնում է դատարկ տող:

Բացի հաստատունից, ծրագրավորման մեջ կարևորվում է նաև **փոփոխականի** հասկացությունը:

Այն մեծությունները, որոնց արժեքները ծրագրի կապարման ընթացքում կարող են փոփոխվել, կոչվում են փոփոխականներ:

Փոփոխականները ներկայացվում են իդենտիֆիկատորներով, որոնք հանդիսանում են դրանց անվանումները: *Յուրաքանչյուր փոփոխականի անունն եզակի է* և ծրագրի կատարման ընթացքում *չի կարող փոփոխվել*: Փոփոխականը ծրագրում կիրառելու համար նախապես պետք է **հայտարարել**, այսինքն՝ ոչ միայն նշել իդենտիֆիկատորը (փոփոխականի անունը), այլև դրա տիպը՝ ամբողջ, իրական, սիմվոլային և այլն, որը բնորոշում է տվյալ փոփոխականի ընդունելիք հնարավոր արժեքների տիպը: Այսպիսով, համակարգիչը, ըստ նշված տիպի, տվյալ փոփոխականին համապատասխան ծավալով հիշողություն է հատկացնում: Օրինակ, եթե կատարվել է *int a=100*; հայտարարությունը, ապա նշանակում է, որ հիշող սարքում *a* փոփոխականին ոչ միայն կոնկրետ ծավալով հիշողություն է հատկացվել, այլև դրան նախնական *100* արժեք է տրվել (նկ. 2.4):



Նկ. 2.4. Փոփոխականի տեղակայումը հիշող սարքում

Այսպիսով, փոփոխականը ծրագրային մոդուլի մեջ մեկ անգամ հայտարարվելով՝ կարող է բազմաթիվ անգամ կիրառվել, արժեքը փոփոխել և այլն: Վերը բերված օրինակում *a*-ն սկզբնարժեքավորվել է (*a = 100*), որը չի խանգարում, որ հետագայում այն այլ արժեքներ կրի: Օրինակ,

$$a = -60;$$

հրամանից հետո *a*-ին տրված հիշողության տիրույթում *100*-ի փոխարեն կգրվի *-60* արժեքը: *a = -60* արտահայտությունը **վերագրման գործողություն** է. վերագրման գործողության ձախ մասում գրվում է այն փոփոխականի անունը, որը պետք է արժևորվի, իսկ աջ մասում՝ արտահայտություն, որի արժեքը պետք է տրվի ձախ մասում առկա փոփոխականին: Ընդ որում՝ **վերագրման գործողությունն արժեք ունի**,

որը հավասար է աջ մասում գրված արտահայտության արժեքին: Օրինակ, եթե կատարվել է $int a, b, c=3$; հայտարարությունը (որի համաձայն c -ն ստացել է նախնական 3 արժեքը), ապա $a=b=c+5$; վերագրման գործողության արժեքը հավասար է 8-ի ($b=3+5$), որն էլ վերագրվում է a -ին. այսպիսով, a -ն և b -ն ստանում են աջ մասում գրված $c+5$ արտահայտության արժեքը: Վերագրման գործողության գրառման այլ տեսքերի հետ կծանոթանանք քիչ անց:

$C++$ -ում կիրառում են **ունար** և **բինար** գործողություններ:

Ունար են այն գործողությունները, որոնք միայն մեկ օպերանդի հետ են աշխատում. թվարկենք դրանք՝

- **&** – օպերանդի հասցեն ստանալու գործողություն,
- ***** – հասցեի միջոցով օպերանդի արժեքին դիմելու գործողություն,
- **--** – օպերանդի նշանը հակառակ նշանի փոխելու գործողություն,
- **+** – օպերանդի դրական լինելը փաստող գործողություն,
- **~** – ամբողջթվային արգումենտի բիթային հակադարձում (1 -ը 0 -ի, և 0 -ն՝ 1 -ի),
- **!** – օպերանդի տրամաբանական բացասում (*true-false*, *false-true*),
- ինկրեմենտ կամ **++** – օպերանդի արժեքի ավելացում 1 -ով,
- դեկրեմենտ կամ **--** – օպերանդի արժեքի նվազեցում 1 -ով,
- տիպի ձևափոխման գործողություն,
- *sizeof* գործողություն,
- **::** – տեսանելիության տիրույթի ցուցման գործողություն,
- *new* – հիշողության դինամիկ բաշխման գործողություն,
- *delete* – դինամիկ բաշխված հիշողության ազատում:

Թվարկած գործողությունների մի մասին կծանոթանանք հետագայում: Գիտարկենք **ինկրեմենտ** և **դեկրեմենտ** գործողությունների աշխատանքը:

Այս գործողությունների արգումենտները կարող են լինել թե՛ իրական և թե՛ ամբողջ թվային արժեքներ կրող փոփոխականները: Ինկրեմենտի դեպքում փոփոխականի ընթացիկ արժեքն ավելացվում է 1 -ով: Օրինակ՝ եթե $a=3$; , ապա $a++$; գործողությունից հետո a -ի արժեքը կստացվի՝ $a=4$ և, եթե $b=2.3$, ապա $b++$;-ից հետո կունենանք $b=3.3$:

Դեկրեմենտի դեպքում ($--$) փոփոխականի ընթացիկ արժեքը պակասեցվում է մեկով: Օրինակ՝ եթե $c=5$; , ապա $c--$; -ի արդյունքում կստացվի $c=4$ և, եթե $d=4.75$; , ապա $d--$; -ից հետո կունենանք $d=3.75$:

Տարբերում են ինկրեմենտ և դեկրեմենտ գործողությունների **նախդիրային** ($++a$; $--a$) և **վերջդիրային** ($a++$; $a--$) տարբերակները: Մրանք համարժեք են, եթե կիրառվում են առանձին օպերանդների նկատմամբ, ինչպես բերված օրինակներում, սակայն, եթե կիրառվում են վերագրման գործողության մեջ՝ արդյունքը տարբեր է. աջ մասի արտահայտության արժեքը հաշվելիս նախ իրականացվում է նախդիրային ինկրեմենտի (դեկրեմենտի) գործողությունը և ապա ստացվածը ներառվում արտահայտության վերջնական արժեքը հաշվելու մեջ, իսկ վերջդիրային տարբերակի դեպքում արտահայտության արժեքը հաշվվում է առանց նշված ինկրեմենտի (դեկրեմենտի) գործողությունն իրագործելու, և վերագրման գործողությունն ավարտելուց հետո միայն իրագործվում է վերջդիրային ինկրեմենտը (դեկրեմենտը):

Այսպես, օրինակ՝ ենթադրենք, $a=3$; . այս դեպքում $c=a++$; վերագրման արդյունքում կունենանք $c=3$ և $a=4$: Իսկ $c=++a$; գործողության արդյունքում՝ $c=4$ և $a=4$:

Երբեմն արտահայտության արժեք հաշվելիս անհրաժեշտ է լինում դրա մեջ եղած փոփոխականի (օպերանտի) տիպը նախօրոք փոփոխել, բերել **նպատակային** տիպի. սա նշանակում է, որ առանց համակարգչում օպերանտի տիպը փոփոխելու՝ դրա արժեքն օպերատիվ հիշողությունում կփոխակերպվի անհրաժեշտ նպատակային տիպի:

Տիպի բացահայտ փոխակերպման հրամանն ընդհանուր դեպքում կունենա

(*նպատակային տիպ*) *օպերանդ*;

տեսքը: Օրինակ, $double\ c=(double)2$; արտահայտության արդյունքում նախ 2 ամբողջին օպերատիվ հիշողությունում կտրամադրվի 8 բայթ ծավալով տիրույթ (նախկին 2 բայթի փոխարեն, որը զբաղեցնում էր որպես *int* տիպի հաստատուն), ապա փոխակերպվելով իրական *double* տիպի թվի (2.0) տեսքի՝ կվերագրվի *c*-ին (նշենք, որ $double\ c=2$; արտահայտության իրագործման նպատակով համակարգիչը կատարում է տիպի **անբացահայտ** ձևափոխություն առանց որևէ հրահանգի):

Նման փոխակերպումների դեպքում հաճախ (օրինակ, եթե իրական մեծությունը փոխակերպվում է ամբողջ տիպի) իմաստալից թվանշանների կորուստ կարող ենք ունենալ, եթե թվի նախնական մեծությունն ավելին է, քան բերվող տիպին տրամադրվող հիշողությունն է թույլատրում: Այս դեպքում արդյունքն անորոշ է: Տիպի փոխակերպման գործընթացում սխալից խուսափելու համար հաճախ օգտվում են հետևյալ առավել անվտանգ միջոցներից՝

w) dynamic_cast<նպատակային տիպ> արտահայտություն,

p) static_cast<նպատակային տիպ> արտահայտություն:

Առաջին (ա) դեպքում ծրագրի իրագործման ընթացքում նախ ստուգվում է տիպի փոխակերպման արդյունքում հնարավոր սխալի առկայությունը, և եթե նման վտանգ չկա, տիպի ձևափոխումն իրագործվում է, հակառակ դեպքում՝ ոչ:

Երկրորդ (բ) դեպքում տիպի ձևափոխման թույլատրելիությունն ստուգվում է ծրագրի թարգմանման փուլում:

sizeof գործողությունն կիրառվում է օպերանտի գրաված հիշողության ծավալը (բայթերով) որոշելու համար: Ընդ որում՝ կիրառելի են *sizeof* գործողության հետևյալ երկու տարբերակները՝

w) sizeof արտահայտություն,

p) sizeof (տիպ):

Օրինակ՝ $sizeof(double)$, $sizeof(a+7.8)$:



1. Թվարկեք C++-ի գրելաչև չունեցող պայմանանշանները:

2. Ի՞նչ է լեկսեմը. ինչպե՞ս է այն կազմվում:

3. Ինչպե՞ս են կազմվում իդենտիֆիկատորները:

4. Ստորև բերված իդենտիֆիկատորներից ընտրեք ճիշտ կազմվածները.

- | | | |
|----------|----------|----------|
| ա) bool | բ) _1c | գ) d ab |
| դ) k+3 | ե) 5mm | զ) abc |
| է) a-b | ը) c_1 | թ) -k |
| ժ) 12 | ի) Mike | լ) Levon |
| խ) Արմեն | ծ) So_na | |

5. Քանի՞ տիպի հաստատություններ գիտեք:

6. Ինչպե՞ս են սրեղծվում քասնվեցական հաստատությունները:

7. Հիշողության քանի՞ բայթ է փրամադրվում հեղեղայ սմբողջ տիպի հաստատություններից յուրաքանչյուրին.

- | | | |
|----------|---------|---------|
| ա) char, | բ) int, | գ) long |
|----------|---------|---------|

8. Ստորև բվարկածներից որո՞նք են ճիշտ սիմվոլային հաստատություններ.

- | | | |
|-----------|--------|---------|
| ա) "true" | բ) 'a' | գ) '\n' |
| դ) "m" | ե) '1' | զ) '\t' |

9. Ի՞նչ է փրամաքանական հաստատությունը, քանի՞ սրժեր կարող է այն ընդունել:

10. Տողային հաստատունի որևէ օրինակ բերեք:

11. Ո՞ր մեծություններն են կոչվում փոփոխական:

12. Ինչպե՞ս են հայտարարվում փոփոխականները:

13. Ինչպե՞ս է հաշվարկվում վերագրման գործողության արժեքը՝

ա) չսխից աջ,

բ) աջից չսխ:

14. C++-ում քանի՞ տիպի գործողություններ գիտեք:

15. Ի՞նչ է ինկրեմենտը:

16. Ի՞նչ է դեկրեմենտը:

17. Ինչպե՞ս են double տիպի փոփոխականը բացահայտ փոխակերպում int տիպի:

18. Ի՞նչ է վերադարձնում sizeof գործողությունը:

§ 2.3

ԲԻՆԱՐ ԳՈՐԾՈՂՈՒԹՅՈՒՆՆԵՐ: ԹՎԱԲԱՆԱԿԱՆ ԵՎ ՏՐԱՍԲԱՆԱԿԱՆ ԱՐՏԱՀԱՅՏՈՒԹՅՈՒՆՆԵՐ: ՕՊԵՐԱՏՈՐ:

Այժմ ծանոթանանք C++-ում կիրառելի բինար գործողություններին:

Բինար գործողությունները կարելի է համախմբել ըստ հետևյալ գործողությունների.

- ադդիտիվ (գումարային),
- մուլտիպլիկատիվ (բազմապատկման),
- տեղաշարժի,
- կարգային,
- համեմատման,
- տրամաբանական,
- վերագրման,
- տեսանելիության տիրույթի թույլատրելիության,
- ստորակետի:

Ծանոթանանք սրանցից յուրաքանչյուրին:

Ադդիտիվ գործողությունները գումարման (+) և հանման (-) գործողություններն են, որոնք իրագործվում են թվաբանությունից հայտնի օբեկտներով:

Մուլտիպլիկատիվ գործողությունները բաժանման և բազմապատկման համար նախատեսված հետևյալ գործողություններն են.

- * – թվային օպերանտների բազմապատկում,
- / – թվային օպերանտների բաժանում. այս բաժանման առանձնահատկությունն այն է, որ ամբողջ տիպի մեծություններն իրար վրա բաժանելիս քանոթի հնարավոր կոտորակային մասը դեն է նետվում. արդյունքում նորից ամբողջթվային արժեք է ստացվում: Օրինակ՝

$$10/3=3, \quad -10/3=-3, \quad 10/(-3)=-3:$$

- % – ամբողջ թվերը բաժանելու արդյունքում ամբողջթվային մնացորդի ստացում: Այս բաժանման արդյունքը միշտ ամբողջ թիվ է, որի նշանը համընկնում է բաժանելիի նշանի հետ: Օրինակ՝

$$\begin{array}{ll} 10\%3=1 & (10:3=3 \text{ և } 1 \text{ մնացորդ}), \\ 6\%4=2 & (6:4=1 \text{ և } 2 \text{ մնացորդ}), \\ -6\%4=-2 & (\text{նշանը համընկնում է բաժանելիի } (-6) \text{ նշանի հետ}), \\ 6\%(-4)=2 & (\text{նշանը համընկնում է բաժանելիի } (6) \text{ նշանի հետ}), \\ -6\%(-4)=-2 & (\text{նշանը համընկնում է բաժանելիի } (-6) \text{ նշանի հետ}): \end{array}$$

Տեղաշարժի գործողությունները սահմանված են միայն ամբողջթվային օպերանտների համար: Ընդհանուր տեսքն այսպիսին է.

$$\text{օպերանդ1} \quad \text{տեղաշարժի գործողություն} \quad \text{օպերանդ2};$$

որտեղ *տեղաշարժի գործողությունը* կամ << է (երկու իրար հաջորդող փոքրի նշանները՝ միջնամասում առանց բացատանիշի), կամ >> (երկու իրար հաջորդող մեծի նշաններ):

<< գործողության արդյունքում *օպերանդ1*-ի պարունակությունը (երկուական կոդը) տեղաշարժվում է ձախ, իսկ >> գործողության արդյունքում՝ դեպի աջ՝ *օպերանդ2*-ին հավասար քանակությամբ: Օրինակ, եթե *int c=7;* , ապա *c<<5;* հրամանի արդյունքում կստանանք հետևյալը՝

0	0	0	0	0	0	0	0	0	0	0	0	0	7		
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1

C-ն մինչև շեղումը

0	0	0	0	0	0	0	3	4	0						
0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0

C-ն շեղումից հետո

Ինչպես կարելի է տեսնել՝ $(7)_8$ -թիվը 5 դիրքով դեպի ձախ շեղելու արդյունքում դարձել է հավասար $(340)_8$, որն էլ 10-ական համակարգի 224 թիվն է՝

$$(340)_8 = (224)_{10} = 7 \times 2^5:$$

Այսպիսով՝

Թիվը k դիրքով դեպի ձախ շեղելը հավասարազոր է այն 2^k -ով բազմապատկելուն:

Միևնույն տրամաբանությամբ՝

k -դիրքով դեպի աջ շեղելու արդյունքում թիվը բաժանվում է 2^k -ի վրա (սրացված մնացորդային մասը դեն նետելով):

Կարգային գործողությունները իրականացվում են ամբողջթվային օպերանդների երկուական կոդերի հետ: Դրանք հետևյալն են՝

- & – կարգային բազմապատկում,
- | – կարգային գումարում (կամ),
- ^ – կարգային բացասող գումարում (բացասող կամ):

Օրինակ՝ $5 \& 6 = 4$, քանի որ $(101 \& 110) = (100) = 4$,
 $5 | 6 = 7$, քանի որ $(101 | 110) = (111) = 7$,
 $5 \wedge 6 = 3$, քանի որ $(101 \wedge 110) = (011) = 3$:

Համեմատման գործողությունները հետևյալն են.

- < փոքր է,
- <= փոքր է կամ հավասար,
- > մեծ է,
- >= մեծ է կամ հավասար,

- == հավասար է (հավասարության ստուգում),
- != հավասար չէ (անհավասարության ստուգում):

Այս գործողությունների արդյունքը *տրամաբանական տիպի* արժեք է՝ *true* (ճիշտ) կամ *false* (սխալ): Նշենք, որ ($=$) և ($!=$) գործողությունների իրագործման կարգն ավելի ցածր է, քան մնացած համեմատման գործողությունների կարգը: Այսպիսով, օրինակ,

$$(c < 3 == 1 < c)$$

արտահայտության արժեքը միայն այն դեպքում կլինի *true*, երբ տեղի ունի $1 < c < 3$ պայմանը, հակառակ դեպքում կլինի *false* (նախ ստուգվում են $c < 3$ և $c > 1$ պայմանները, ապա ստացված տրամաբանական արժեքները համեմատվում հավասարության առումով):

Քինար տրամաբանական գործողությունները երկուական են. ա) *&&* – *կոնյունկցիա* (տրամաբանական *և* և *բ*) *||* – *դիզյունկցիա* (տրամաբանական *կամ*):

Սրանց արդյունքը նույնպես տրամաբանական արժեք է՝ *true* կամ *false*: Ընդ որում՝

Կոնյունկցիայի արդյունքը true է միայն այն դեպքում, երբ քինար գործողությանը մասնակից երկու օպերանդներն էլ true արժեք ունեն: Դիզյունկցիայի արդյունքը true է, եթե օպերանդներից թեկուզ մեկի արժեքը true է:

Սսենք, որ արտահայտության արժեքը հաշվելիս *&&* և *||* գործողությունների կատարման առաջնահերթության աստիճանը համեմատման գործողությունների համեմատ ավելի ցածր է, օրինակ՝ $4 != 3 || 5 > 7$ արտահայտության արժեքը *true* է, քանի որ $4 != 3$ -ի արդյունքը *true* է՝ թեպետ $5 > 7$ -ինը *false* է, իսկ $true || false = true$: Հիշեցնենք, որ *true*-ի թվային արժեքը 1 է, իսկ *false*-ինը՝ 0:

Վերագրման գործողության հնարավոր տեսքերից մեկին արդեն ծանոթ եք՝ $A=B$, որի արդյունքում A -ի նախկին արժեքը փոխարինվում է B -ի արժեքով: Այստեղ որպես A կարող է հանդես գալ միայն փոփոխականը (իդենտիֆիկատորը), իսկ որպես B ՝ ցանկացած հաստատուն, փոփոխական կամ արտահայտություն, որի արժեքը A -ի տիպի է (կամ տիպի փոխակերպմամբ կարող է ներառվել A -ի հնարավոր արժեքների մեջ):

Ծանոթանանք *վերագրման գործողության* մնացած եղանակներին.

- *= – ձախ մասում գրված օպերանտի արժեքը բազմապատկել այ մասում եղածով, արդյունքը գրել ձախի մեջ,
- /= – ձախ մասում գրված օպերանտի արժեքը բաժանել այ մասում եղածի վրա, արդյունքը գրել ձախի մեջ,
- %= – ձախ մասում գրված ամբողջ օպերանտի արժեքը բաժանել այ ամբողջ օպերանտի վրա և ստացված ամբողջթվային մնացորդը վերագրել ձախ օպերանտին,

- = – ձախ մասի օպերանտից հանել աջ օպերանտի արժեքն ու արդյունքը գրել ձախի մեջ,
- += – ձախ օպերանտին գումարել աջ մասում եղածն ու արդյունքը գրել ձախի մեջ,
- &= – հաշվել ձախ և աջ մասերի ամբողջ օպերանտների *բիթային (կարգային) կոնյունկցիան* և արդյունքը գրել ձախ օպերանտի մեջ,
- |= – հաշվել ձախ և աջ մասերի ամբողջ օպերանտների կարգային դիզյունկցիան և արդյունքը գրել ձախ օպերանտի մեջ,
- ^= – ձախ և աջ մասերի *բացասող կամ* գործողության արդյունքը վերագրել ձախ մասին,
- <<= – ձախ մասի ամբողջթվային արժեքի բիթային ներկայացման *շախտի տեղաշարժ* աջ մասի ամբողջթվային օպերանտի չափով,
- >>= – ձախ մասի ամբողջթվային արժեքի բիթային ներկայացման *աջ տեղաշարժ* աջ մասի ամբողջթվային օպերանտի չափով:

Վերը բերված վերագրման գործողությունները համարժեք են

շախտ օպերանտը = շախտ օպերանտ գործողություն աջ օպերանտ

վերագրման գործողությանը: Օրինակ՝ եթե $a = 3$; ապա $a+ = 2$; և $a = a + 2$; վերագրումների արդյունքները նույն են՝ երկու դեպքում էլ a -ի արժեքը կստացվի 5:

Բացի սահմանված գործողություններից, ցանկացած ծրագրավորման լեզվում կիրառվում են նաև *օպերատորներ*:

Օպերատորները հապուկ չհակերպված հրամաններ (հրահանգներ) են, որոնք ունեն իրենց աշխարհարգերը:

Օպերատորները միմյանցից փոխանջատվում են կետ-ստորակետերով (;). երկու իրար հաջորդող կետ-ստորակետերը ստեղծում են *դասարկ օպերատոր*: Ռատարկ օպերատոր կիրառում են, եթե լեզվի օրենքները տվյալ տեղում օպերատորի առկայություն են պահանջում, մինչդեռ ըստ լուծվող խնդրի այդպիսինի անհրաժեշտություն չկա:

Չևավոր փակագծերի {} միջև առնված օպերատորների հաջորդականությունն անվանում են բաղադրյալ օպերատոր: Եթե բաղադրյալ օպերատորի կազմում կան նաև փոփոխականների հայտարարություններ, ապա այն կազմում է բլոկ:

$C++$ -ի կոմպիլյատորը թե՛ *բաղադրյալ օպերատորը* և թե՛ *բլոկը* դիտում է որպես մեկ ամբողջություն: Ընդ որում՝ ինչպես բաղադրյալ օպերատորը, այնպես էլ բլոկը սահմանափակող } փակագծից հետո կետ-ստորակետ (;) չի դրվում:

Բլոկում սահմանված (հայտարարված) մեծությունները հայտնի են միայն տվյալ բլոկում և դրանից դուրս մատչելի չեն:

Այն մեծությունները, որոնք հայտարարվում կամ սահմանվում են բլոկում, լոկալ (տեղային) են, և դրանց տեսանելիության տիրույթը սահմանափակված է բլոկի տիրույթով:

Ինչպես արդեն գիտենք, ցանկացած ծրագիր պարունակում է **գլխավոր մաս** (`main()`), որի մարմինը պարփակված է ձևավոր փակագծերով, այսինքն՝ բլոկ է կազմում, որտեղ հայտարարված մեծությունները, այսպիսով, լոկալ են և հայտնի միայն `main()` ֆունկցիայի մարմնում (բլոկում): Որպեսզի փոփոխականներն ու սահմանվող մեծությունները մատչելի դառնան ծրագրի ողջ տարածքում, դրանք պետք է հայտարարվեն `main()`-ից և ցանկացած այլ ֆունկցիայից դուրս: Այսպիսի մեծություններն անվանում են **գլոբալ**: Գլոբալ մեծություններին հաճախ դիմում են **:: (պատկանելիություն) ունար գործողության** միջոցով: Օրինակ, եթե `d` անվամբ փոփոխական ունենք հայտարարված թե՛ `main()`-ում և թե՛ դրանից դուրս, ապա միայն **::** գործողությամբ է հնարավոր `main`-ից դիմել և աշխատել գլոբալ (արտաքին) `d` փոփոխականի հետ.

```
#include <iostream.h>
int d=3;
void main ()
    { int d=10; d++;
      cout << d << endl;           //1
      :: d+=5;
      cout << :: d < endl;         //2
    }
```

Այս ծրագրի կատարման արդյունքում նախ `//1` տողով կարտածվի `11` թիվը, որը ստացվում է լոկալ `d`-ի ինկրեմենտի արդյունքում, իսկ `//2` տողում՝ `8` թիվը, որը ստացվում է գլոբալ `d`-ի արժեքին (3) ավելացնելով 5: Այսպիսով, թեպետ այստեղ երկու փոփոխականներ կրում են միևնույն անունը (`d`), սակայն դրանք իրարից անկախ, տարբեր արժեքներ կրող մեծություններ են:

Ստորակետը ծառայում է նաև որպես գործողություն, ընդ որում՝ ստորակետերով միմյանցից բաժանված արտահայտությունները հաշվվում են հաջորդաբար՝ ձախից աջ: Այսպիսով, **ստորակետը** գործողությունը խմբավորում է արտահայտություններն այնպես, որ ստացվող արդյունքի տիպն ու արժեքը որոշվում են արտահայտության աջ մասում եղած արտահայտությամբ, ձախ մասում եղած օպերանտների արժեքներն անտեսվում են:

Օրինակ՝ `cout << (k=5, k+1)`; հրամանի արդյունքում էկրանին կարտածվի `6` թիվը (`k+1`-ի արժեքը՝ երբ `k=5`): Եթե այժմ իրագործենք `cout << k`; հրամանը՝ արդյունքում կտեսնենք `5` թիվը:

Պայմանական կամ տերնար գործողություն

Ի տարբերություն բինարի, **պայմանական (տերնար)** գործողությունը երեք օպերանտ ունի. այն հետևյալ ընդհանուր տեսքի է.

1-ին արտահայտություն ? 2-րդ արտահայտություն: 3-րդ արտահայտություն

Այս գործողության կատարումը սկսվում է *1-ին արտահայտության* արժեքի որոշմամբ. եթե այն հավասար է *true* (հավասար չէ 0), ապա իրականացվում է *2-րդ արտահայտությունը*, հակառակ դեպքում՝ *3-րդ արտահայտությունը*:

Այսպիսով, պայմանական (տերնար) գործողության արդյունքն իրագործված (1-ին կամ 2-րդ) արտահայտության արժեքն է: Օրինակ՝ $y = x < 0 ? -x : x$; արտահայտության արժեքը կլինի x -ի մոդուլը՝ բացարձակ արժեքը:

Թվաբանական արտահայտությունները կազմվում են *գումարման (+), հանման (-), բազմապատկման (*), բաժանման (/)* և *բաժանման ամբողջ մնացորդի առանձնացման (%)* գործողությունների ու *սրահարյուր մաթեմատիկական ֆունկցիաների* կիրառմամբ: Ստանդարտ մաթեմատիկական ֆունկցիաներից (հավելված 2) օգտվելու համար անհրաժեշտ է `#include <math.h>` դիրեկտիվի (հրահանգ) միջոցով ծրագրին կցել այդ ֆունկցիաների նկարագրությունները պարունակող `math.h` վերնագրային ֆայլը: Թվաբանական արտահայտության արժեքը հաշվելիս պետք է առաջնորդվել գործողությունների կատարման առաջնահերթության կանոններով, որոնք ըստ էության համընկնում են հանրահաշվում ընդունված կանոնների հետ. նախ իրագործվում են () փակագծերում ներառված գործողությունները, ապա՝ ձախից այդ ըստ դրանց գրառման հաջորդականության *, /, և % գործողությունները, իսկ վերջում՝ գումարման ու հանման:

Օրինակ՝ եթե անհրաժեշտ է հաշվել $\frac{x + y + z}{3}$ արտահայտության արժեքը, ապա այն պետք է գրել հետևյալ կերպ՝ $(x + y + z) / 3$, ընդ որում՝ եթե այս գրառման փոխարեն կիրառենք $x + y + z / 3$ արտահայտությունը, ապա վերջինս պետք եղածի փոխարեն կհաշվի $x + y + \frac{z}{3}$ արտահայտության արժեքը:

Տրամաբանական արտահայտությունները կազմվում են *համեմատման* և *տրամաբանական* գործողությունների համակցմամբ: Տրամաբանական արտահայտության արժեքը *true* կամ *false* է: Օրինակ՝ տերնար գործողության ձախ մասում (մինչև ? -ը) գրվում է տրամաբանական արտահայտություն՝

$$a > b ? m = a : m = b;$$

որտեղ $a > b$ համեմատման գործողությունը տրամաբանական արտահայտություն է, որը a -ի և b -ի կոնկրետ արժեքների դեպքում *true* կամ *false* արժեք կստանա:



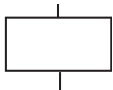
1. Ի՞նչ քինար գործողություններ գիտեք:
2. Որո՞նք են աղդիտիվ գործողությունները:
3. ԹՎարկեր մուլտիպլիկատիվ գործողությունները:
4. Ի՞նչ տիպի արժեք է վերադարձնում % գործողությունը՝
 - ա) ամբողջ,
 - բ) իրական:
5. Տեղաշարժի ի՞նչ գործողություններ գիտեք:
6. Ինչի՞ է հավասարագործ 8-ական ամբողջ թիվը 3 դիրքով դեպի ձախ տեղաշարժելը:
7. Ի՞նչ տիպի արժեք է ստացվում համեմատման գործողությունների արդյունքում:
8. Ի՞նչ է դիվիզիոնիցիան:
9. Ի՞նչ է կոնյունկցիան:
10. Վերագրման հնարավոր մի քանի գործողությունների օրինակ բերեք:
11. Ի՞նչ է բաղադրյալ օպերատորը: Ո՞ր բաղադրյալ օպերատորն են համարում բլոկ:
12. Որո՞նք են կոչվում լոկալ մեծություններ և որտե՞ղ են դրանք հայտնի:
13. Ո՞ր մեծություններն են կոչվում գլոբալ և որտե՞ղ են դրանք հայտարարվում:
14. Ո՞ր գործողության միջոցով են հիմնականում դիմում գլոբալ մեծությանը:
15. Ինչպե՞ս են հաշվարկվում ստորակետերով բաժանված արտահայտությունները:
16. Տեղնար գործողության որևէ օրինակ բերեք:
17. Ո՞րն է կոչվում դասարկ օպերատոր:
18. ԹՎաբանական արտահայտության արժեք հաշվելիս գործողությունների կատարման ի՞նչ առաջնահերթություն է սահմանված:

§ 2.4 ԱԼԳՈՐԻԹՄՆԵՐ

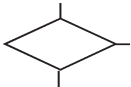
9-րդ դասարանից ծանոթ եք **ալգորիթմ** հասկացությանն ու դրա նկարագրման եղանակներին: Հակիրճ վերհիշենք ալգորիթմների մասին անցած նյութը:

Ալգորիթմը գործողությունների կարգավորված հաջորդականություն է, որը հանգեցնում է սպասված արդյունքին:

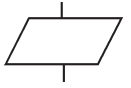
Ալգորիթմ նկարագրելու տարբեր եղանակներից ծանոթ ենք *քառաքանաչևային* և *գրաֆիկական* եղանակներին: Քանի որ օգտվելու ենք գրաֆիկական ներկայացումից՝ վերհիշենք դրա տարրերը.



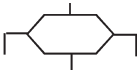
հաշվարկների կատարման և վերագրման գործողություն,



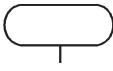
պայմանի ստուգում և հաշվման գործընթացի այլընկրան-քային շարունակում,



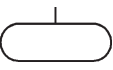
տվյալների ներմուծում, տվյալների արտաձույթ,



ցիկլային գործընթացի կազմակերպում,



ալգորիթմի սկիզբ,



ալգորիթմի ավարտ,



ալգորիթմի հոսքի ընդհատված մասերի կապի միջոց:

Ալգորիթմները՝ կախված տվյալ պահին լուծվող խնդրից, կարող են լինել **գծային**, **ճյուղավորված** և **ցիկլային**:

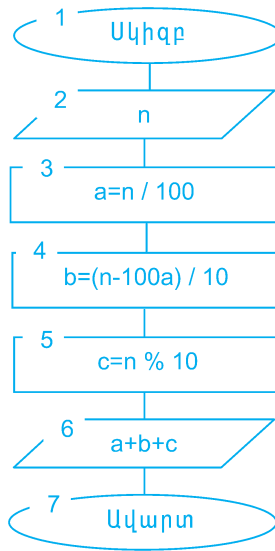
Գծային են կոչվում այն ալգորիթմները, որտեղ, պարամետրերի արժեքներից անկախ, գործողությունները կատարվում են միշտ միևնույն հաջորդականությամբ՝ վերից վար, յուրաքանչյուրը՝ միայն մեկ անգամ:

Գիտարկենք հետևյալ խնդիրը.

Տրված է եռանիշ n թիվը: Պահանջվում է հաշվել թիվը կազմող բաղադրիչ թվանշանների գումարը:

Նախ կազմենք խնդրի լուծման բլոկ-սխեման (նկ. 2.5), ապա՝ ծրագիրը:

Բերված ալգորիթմում կիրառվել են ամբողջ թվերի համար սահմանված / և % գործողությունները, որտեղ /-ը վերադարձնում է երկու ամբողջ թվերի բաժանումից ստացվող քանորդի ամբողջ արժեքը (օրինակ՝ $7 / 3 = 2$), իսկ %-ը՝ այդ բաժանման արդյունքի ամբողջ մնացորդը (օրինակ՝ $7 \% 3 = 1$):



Նկ. 2.5. Եռանիշ թվի թվանշանների գումարի հաշվման ալգորիթմ

Եթե, օրինակ, $n = 672$, ապա 3-րդ բլոկով կստանանք $a = 672 / 100 = 6$, որը հարյուրավորն է, 4-րդ բլոկով՝ $b = (672 - 100 \cdot 6) / 10 = 7$, որը տասնավորն է, իսկ 5-րդով կստանանք $c = 672 \% 10 = 2$, որը միավորն է:

Այսպիսով, a , b , c փոփոխականների մեջ ստացվել են եռանիշ թվի բաղադրիչ թվանշանները, մնում է 6-րդ բլոկով արտածել պահանջվող գումարը:

Կազմենք ծրագիրը.

```

#include <iostream.h>
void main ()
{ int n;
  int a,b,c;           //a-ն հարյուրավորի, b-ն տասնավորի, c-ն միավորի համար է
  cin >> n;
  a=n/100;             // հարյուրավորի սրացում
  b=(n - 100 * a) / 10; // տասնավորի սրացում
  c=n % 10;           // միավորի սրացում
  cout<<a+b+c<<endl; //1
}
  
```

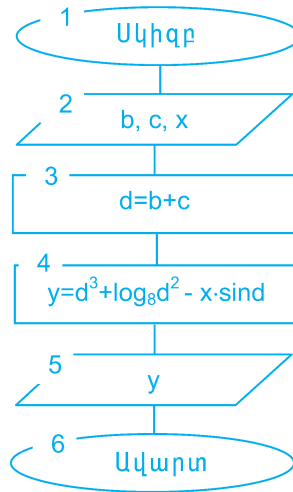
Այստեղ `//1 տողում առկա cout<<a+b+c<<endl;` գրառման մեջ `<<endl`-ի կիրառման արդյունքում պատասխանն արտածելուց հետո կուրսորը կանցնի հաջորդ տող:

Ինչպես երևում է վերը բերված ծրագրից, գծային ալգորիթմների օգնությամբ լուծվող խնդիրները կարող են պարունակել միայն ներմուծման, արտածման հրամաններ և հաշվարկներ կատարելու համար՝ վերագրման օպերատորներ:

Գիտարկենք գծային ալգորիթմով լուծվող ևս մի խնդիր.

x , b , c պարամետրերի ցանկացած իրական արժեքների համար հաշվել և արտածել y -ի արժեքը, եթե $y = (b + c)^3 + \log_8(b + c)^2 - x \sin(b + c)$:

Նախ կառուցենք խնդրի լուծման բլոկ-սխեման (նկ. 2.6).



Նկ. 2.6. $y = (b + c)^3 + \log_8(b + c)^2 - x \sin(b + c)$
 արտահայտության հաշվման ալգորիթ

Բերված ալգորիթմի 3-րդ բլոկում լրացուցիչ d փոփոխականի մեջ պահվել է $b+c$ արտահայտության արժեքը, որպեսզի 4-րդ բլոկում ներառված արտահայտության արժեքը հաշվարկելիս նույն արժեքը $(b+c)$ մի քանի անգամ չհաշվենք: Գրենք ծրագիրը.

```

#include <iostream.h>
#include <math.h> //1
void main ( )
{
    double d,b,c,x,y;
    cout << "b="; cin >> b;
    cout << "c="; cin >> c;
    cout << "x="; cin >> x;
    d=b+c ;
    y=pow(d,3)+ log(pow(d,2)) / log(8) - x * sin(d); //2
    cout << "y=" << y << endl;
}
  
```

//1 տողում *math.h* վերնագրային ֆայլի կցումն անհրաժեշտ է, քանի որ ծրագրում կիրառել ենք դրանում սահմանված մի շարք ֆունկցիաներ՝ *pow*, *log* և *sin*: Ընդ որում՝ $pow(a,b)$ ֆունկցիան վերադարձնում է a -ի b աստիճանը, $log(a)$ -ն՝ a -ի բնական հիմքով լոգարիթմը, իսկ $sin(a)$ -ն՝ ռադիաններով արտահայտված a անկյան սինուսը:

$\log_8 b^2$ արտահայտության արժեքը հաշվելու համար կիրառվել է $\log_8 d^2 = \frac{\log d^2}{\log 8}$ բանաձևը, որով լոգարիթմի 8 հիմքից անցում է կատարվել e բնական հիմքին (*math.h*

Ֆայլում արգումենտի e բնական հիմքով լոգարիթմը հաշվող ստանդարտ ֆունկցիան կրում է \log անվանումը): *math.h* ֆայլում սահմանված է նաև $\log10(x)$ ֆունկցիան, որը վերադարձնում է լոգարիթմ տաս հիմքով x -ի արժեքը:



1. **Գծային ալգորիթմները ծրագրավորելիս n^2 օպերատորներն են կիրառվում:**
2. **Էկրանին ավյալներ արտածելիս հաջորդ պողին անցում կատարելու ի՞նչ միջոց գիտեք:**
3. ***math.h* սրանդարտ գրադարանային ֆայլում սահմանված ի՞նչ ֆունկցիաներ գիտեք:**
4. **Կազմեք հետևյալ խնդիրների լուծման բլոկ-սխեմաներն ու ծրագրերը.**
 - **Հաշվել և արտածել արված քառանիշ թվի թվանշանների արտադրյալը:**
 - **Տրված եռանիշ թվի մեջ տեղերով փոխել միավորների և տասնավորների թվանշանների տեղերը: Արտածել սրացված նոր եռանիշ թիվը:**
 - **Տրված քառանիշ թվի մեջ տեղերով փոխել միավորների և հազարավորների, տասնավորների և հարյուրավորների թվանշանների տեղերը: Արտածել սրացված նոր քառանիշ թիվը:**
 - **Օգտվելով հավելված 2-ում բերված սրանդարտ գրադարանային ֆունկցիաներից, x -ի ցանկացած իրական արժեքի համար հաշվել և արտածել y -ի արժեքը, եթե՝**
 - ա) $y = (x + 1) (x^2 + 1)^2 \sin(x + 3) \operatorname{tg}(x)$,
 - բ) $y = \frac{x - 4}{x^2 + 2} + 2^x$,
 - գ) $y = \operatorname{ctg} \frac{x}{|x| + 1} + \lg(x^2 + 1)$,
 - դ) $y = \ln(e^x + 1) + \sqrt[3]{x + 2}$.

ՃՅՈՒՂԱՎՈՐՄԱՆ ԳՈՐԾԸՆԹԱՑ:

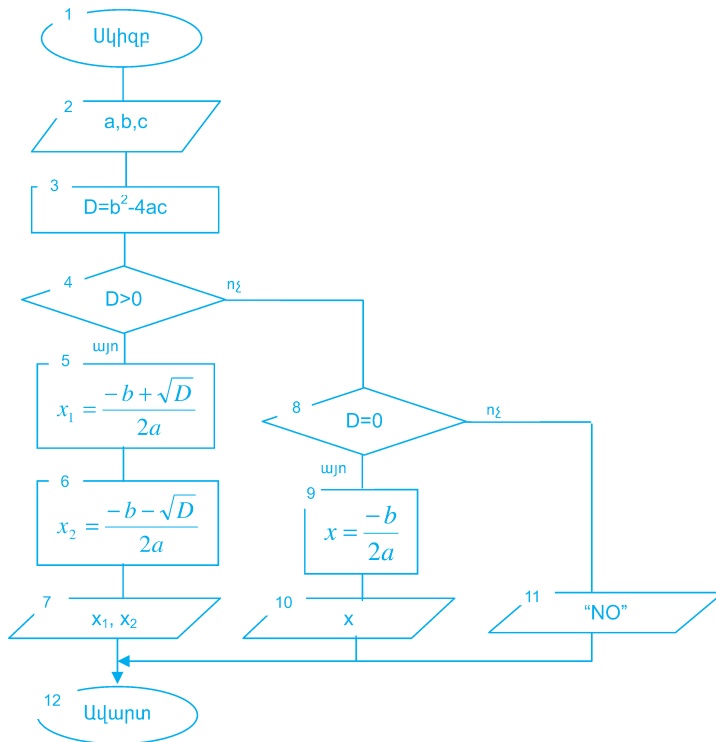
§ 2.5

ՃՅՈՒՂԱՎՈՐՄԱՆ (ՊԱՅՄԱՆԻ) ՕՊԵՐԱՏՈՐՆԵՐ: ԱՆՊԱՅՄԱՆ ԱՆՑՄԱՆ ՕՊԵՐԱՏՈՐ

Հաճախ խնդիրների լուծման ալգորիթմները, ի տարբերություն գծայինի, **ճյուղավորումներ** են պարունակում. դա բխում է լուծման մեջ առկա պայմաններից, որոնցից կախված խնդրի հետագա լուծումը շարունակվում է տարբեր ճանապարհներով:

Հիշենք, որ այն ալգորիթմը, որտեղ ստուգվող պայմանից կախված խնդրի լուծման գործընթացը շարունակվում է տարբեր ուղիներով, անվանում են **ճյուղավորված**, իսկ համապատասխան ուղիները՝ **ճյուղեր**:

Դիտարկենք $ax^2+bx+c=0$ քառակուսի հավասարման ($a \neq 0$) իրական արմատները փնտրելու ալգորիթմը:



Նկ. 2.7. Քառակուսի հավասարման իրական արմատները փնտրելու ալգորիթմ

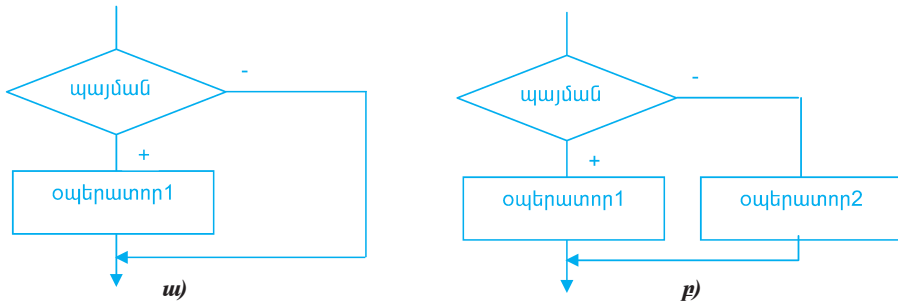
Բլոկ-սխեմայից (նկ. 2.7) երևում է, որ 4-րդ բլոկում ներառված պայմանի ճշմարիտ (*true*) կամ կեղծ (*false*) լինելուց կախված՝ խնդրի լուծման ընթացքը շարունակվում է տարբեր ուղղություններով. ընդ որում՝ պայմանի կեղծ լինելու դեպքում ըստ 8-րդ բլոկի պայմանի ընդունված արժեքի՝ ալգորիթմի մեջ մեկ այլ ճյուղավորում է առաջանում:

Ճյուղավորումներ պարունակող ալգորիթմները ծրագրավորելու համար C++ -ում կիրառում են **պայմանի օպերատոր**, որի հնարավոր տեսքերը հետևյալն են.

ա) *if (a) օպերատոր1;*

բ) *if (a) օպերատոր1; else օպերատոր2;*

որտեղ *a*-ն տրամաբանական կամ թվաբանական արտահայտություն է, իսկ *օպերատոր1*-ը և *օպերատոր2*-ը C++-ի ցանկացած օպերատորներ են կամ { } փակագծերի միջև առնված օպերատորների համախմբություն՝ բաղադրյալ օպերատոր կամ բլոկ: Ընդ որում՝ եթե պայմանի օպերատորը ծրագրավորում է նկ. 2.8 ա)-ում բերված տիպի գործընթաց, ապա կիրառվում է պայմանի օպերատորի **համառոտ** տեսքը՝ *If (a) օպերատոր1;* , հակառակ դեպքում (նկ. 2.8 բ) *if (a) օպերատոր1; else օպերատոր2;* **ընդարձակ տեսքը**:



Նկ. 2.8. Այգորիքների ճյուղավորումը

Այժմ կազմենք **քառակուսի հավասարման արմատների որոշման** (նկ. 2.7) ծրագիրը.

```
#include <iostream.h>
#include <math.h>
void main ()
{ double x1, x2, x, z, d, a, b, c;
  cout << "a="; cin >> a;
  cout << "b="; cin >> b;
  cout << "c="; cin >> c;
  d = pow(b,2) - 4 * a * c ;
  z = 2*a ;
  if (d>0)
  {
    x1 = (- b - sqrt(d)) / z ;
    x2 = (- b + sqrt(d)) / z ;
    cout << "x1=" <<x1 <<endl;
    cout << "x2=" <<x2 <<endl;
  }
  else
  if (d==0) { x=-b/z ;
    cout << "x=" <<x <<endl;
  }
  else cout << "NO" <<endl;
}
```

Ճյուղավորումներով ալգորիթմներ ծրագրավորելիս երբեմն հարմար է պայմանի օպերատորի փոխարեն *ընդհանուր օպերատոր* կիրառել: **Ընդհանուր օպերատորը** հարմար է օգտագործել այն դեպքերում, երբ ճյուղավորված ալգորիթմները ներառված (*if (a1) օպերատոր1; else if (a2) օպերատոր2; else... և այլն*) նոր ճյուղավորումներ են պարունակում:

Այս օպերատորի ընդհանուր տեսքը հետևյալն է.

```
switch (արտահայտություն)
{
    case 1-ին արժեք: օպերատորներ;
    case 2-րդ արժեք: օպերատորներ;
    .
    .
    .
    case n-րդ արժեք: օպերատորներ;
    default : օպերատորներ;
}
```

Այս օպերատորի աշխատանքը հանգում է հետևյալին. նախ հաշվվում է *switch*-ի տակ առնված արտահայտության արժեքը, որը պետք է լինի ամբողջթվային, այնուհետև հերթով, վերից վար ստուգվում է, թե այն *case*-ի ո՞ր արժեքի հետ է համընկնում: Եթե այդպիսի արժեք գտնվում է, ապա իրագործվում են տվյալ *case*-ին հաջորդող օպերատորները: Եթե *switch*-ի արժեքը չի համընկնում *case*-երից ոչ մեկի հետ, իրագործվում են *default*-ի օպերատորները: Սակայն պարտադիր չէ, որ *switch*-ը *default*-ով սկսվող տող ներառի. այս դեպքում, եթե *switch*-ի արժեքը չի համընկնում *case*-ի արժեքներից ոչ մեկի հետ, *switch*-ն ավարտում է աշխատանքը:

Որպեսզի *case*-ի որևէ արժեքին համապատասխանող օպերատորները կատարելուց հետո ավտոմատ չիրագործվեն նաև հաջորդող *case*-երի օպերատորները ևս, անհրաժեշտ է յուրաքանչյուր *case*-ին հաջորդող շարքը ավարտել *break* օպերատորով, որը կխզի *switch*-ի հետագա գործողությունը: *switch*-ի կիրառմանը ծանոթանանք հետևյալ խնդրի միջոցով:

Խնդիր: Ըստ աշակերտի ստացած թվային գնահատականի՝ արտածել դրա բառային հոմանիշը՝ 2-անբավարար, 4-բավարար և այլն:

```
#include <iostream.h>
void main()
{ int n; cin >> n;
  switch (n)
  {
      case 1:
      case 2:
      case 3: cout << "անբավարար"; break;

      //1

      case 4:
      case 5:
      case 6: cout << "բավարար"; break;
```

```

        case 7:
        case 8: cout << "լավ"; break;
    case 9:
    case 10: cout << "գերազանց"; break;
    default : cout << "սխալ գնահատական է ներմուծվել"; break;
}
}

```

Այսպիսով, եթե ներմուծվել է 1, 2 կամ 3 նիշերից որևէ մեկը՝ կարտածվի «անբավարար» բառը և //1 տողի *break* հրամանով *switch*-ի աշխատանքը կավարտվի: Նմանօրինակ գործընթաց կիրականացվի նաև մնացած գնահատականների դեպքում: Սակայն եթե բերված գնահատականներից ոչ մեկի հետ *n*-ի արժեքը չի համընկնում՝ կիրագործվեն *default*-ին հաջորդող օպերատորները (ընդ որում՝ *break*-ն այստեղ կարելի է չգրել):

switch-ը այն եզակի օպերատորներից է, որը *case*-ի մեջ ներառված մի քանի օպերատորները չի պարտադրում առնել ձևավոր փակագծերի մեջ:

Երբեմն անհրաժեշտ է լինում ծրագրի կատարման բնական հաջորդական ընթացքը փոխելով՝ անցում կատարել ծրագրի մեկ այլ հատվածի: Այդ նպատակով կիրառում են *անպայման անցման օպերատորը*:

Անպայման անցման օպերատորն ունի հետևյալ տեսքը.

goto իդենտիֆիկատոր;

որտեղ *իդենտիֆիկատորն* անցման օպերատորը ներառող ֆունկցիայի մարմնում որևէ օպերատորի նշիչ է, այլ խոսքով՝ դրա անվանումը, հասցեն:

Օրինակ՝

```

.
.
.
goto ab;
.
.
.
ab: cout << k;
.
.
.

```

goto օպերատորը կիրառելիս չի կարելի անցում կատարել սկզբնարժեքավորմամբ հայտարարվող մեծությունների «վրայով», սակայն եթե սկզբնարժեքավորվող մեծությունը բլոկի մեջ է՝ կարելի է: Շատ կարևոր է նաև *goto* կիրառելիս հիշել, որ չի կարելի դրսից անցում կատարել բլոկի մեջ, մտնել պայմանի, ընտրության և այլ օպերատորների մեջ ներառված տարածք:

Չնայած երբեմն անհնար է լինում *goto* չկիրառել, այդուհանդերձ, հնարավոր սխալներից խուսափելու համար խորհուրդ է տրվում հնարավորինս դրանից հրաժարվել:

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ Չնայած `switch` օպերատորում `case` և `default` բառերով սկսվող փողերը կարելի է ցանկացած հաջորդականությամբ փեղադրել, այդուհանդերձ, ընթեռնելիության փեսանկյունից `default`-ով սկսվող փողը խորհուրդ է փրվում փեղադրել վերջում:
- ◆ `case`-ի և դրա արժեքի (`case` արժեք) միջև պարպադիր պեղք է բացապանիչ դնել:



1. Քանի՞ հնարավոր փեղք ունի պայմանի օպերատորը, որո՞նք են:
2. Ե՞րբ են կիրառում ընկրության օպերատորը. բերեք այն կիրառելու համար հարմար որևէ իրավիճակի օրինակ:
3. Ի՞նչ փոսպի արժեքներ կարող է ընդունել `switch`-ի արքահայտությունը:
4. Ինչպե՞ս է սվարքվում `switch`-ի աշխարանքը, եթե `case`-ի արժեքներից ոչ մեկի հեղ `switch`-ի արքահայտության արժեքը չի համընկնում:
5. Ե՞րբ իմասք ունի `default` կիրառել:
6. Ի՞նչ փեղի կունենա, եթե `case`-ին հաջորդող օպերատորների մեջ `break` չընդգրկվի:
7. Ընկրության օպերատորն իր իմասքով ձեզ հայտնի ո՞ր օպերատորին է նման:
8. Ծրագրի հեղեղյալ հարվածը փոխարինեք պայմանի համարժեք օպերատորով.

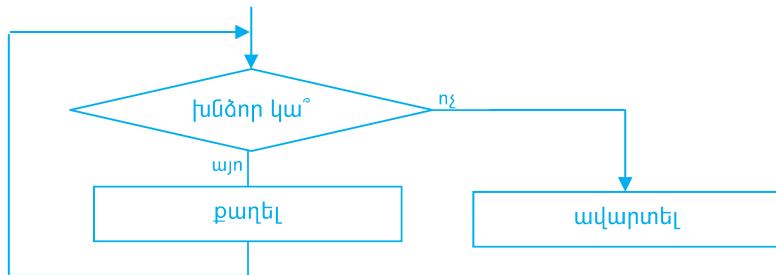

```
.....
switch (k)
{
    case 1: cout<<1; break;
    case 2: cout<<2; break;
}
```
9. Ըսք արեղնաշարից ներմուծված `x`, `y` իրական թվերի և `c` պայմանանշանի արժեքների՝ գրել հեղեղյալ խնդրի լուծման ծրագիրը. եթե `c` պայմանանշանը
 - ‘+’ է՝ հաշվել և արքածել `x + y`-ի արժեքը,
 - ‘-’ է՝ հաշվել և արքածել `x - y`-ի արժեքը,
 - ‘*’ է՝ հաշվել և արքածել `x * y`-ի արժեքը,
 - ‘/’ է, հաշվել և արքածել `x / y`-ի արժեքը,
 իսկ այլ արժեքի դեղքում՝ արքածել “`sxal gorcoxutun e nermucvel`” փեղսքը:
10. Կազմել հավելված 3-ի այս քեմային առնչվող խնդիրների լուծման բլոկ-սխեմաներն ու ծրագրերը:

§ 2.6 ԿՐԿՆՈՒԹՅԱՆ ՕՊԵՐԱՏՈՐՆԵՐ: BREAK ԵՎ CONTINUE ՕՊԵՐԱՏՈՐՆԵՐ

Տարաբնույթ խնդիրներ լուծելիս հաճախ է անհրաժեշտություն ծագում որոշակի գործողությունների իրագործումը կրկնել քանի դեռ դրա անհրաժեշտությունը կա (քանի դեռ որոշակի պայման ճշմարիտ է, այսինքն՝ ունի *true* արժեք):

Օրինակ՝ ռոբոտի համար խնձորենուց բերքի հավաքման գործընթացը կարելի է ձևակերպել հետևյալ կերպ՝ քանի դեռ ծառի վրա խնձոր կա՝ քաղել այն: Այսպիսով, ամեն անգամ խնձոր քաղելուց առաջ ռոբոտը կստուգի գոնե մեկ խնձորի առկայությունը, և եթե այդ պայմանն ընդունի *true* արժեք՝ կկատարի քաղելու գործողություն, հակառակ դեպքում, երբ նշված պայմանը տեղի չունենա, այսինքն՝ այն ստանա *false* արժեք՝ բերքահավաքը կավարտվի:

Նման գործընթացի մեջ խնձոր քաղելը *կրկնվող գործողությունն* է, իսկ գործողության կատարման համար հիմք հանդիսացող պայմանը՝ ծառի վրա խնձորի առկայությունը: Բերված գործընթացը նկարագրենք սխեմատիկորեն.



Ակնհայտ է, որ կգա մի պահ, երբ ծառին այլևս խնձոր չի լինի և քաղելու գործընթացը կավարտվի, բայց եթե, օրինակ, ինչ-որ ձևով քաղելու ընթացքում ծառի վրա նոր խնձորներ «հասցնեին աճել»՝ դժվար է ասել, թե այդ գործընթացն արդյո՞ք ավարտ կունենար: Նման դեպքերում ասում են, որ *անվերջ կրկնողական գործընթաց* կամ *անվերջ ցիկլ* ունենք, որն, իհարկե, բնականոն չէ և որից հնարավորինս պետք է խուսափել:

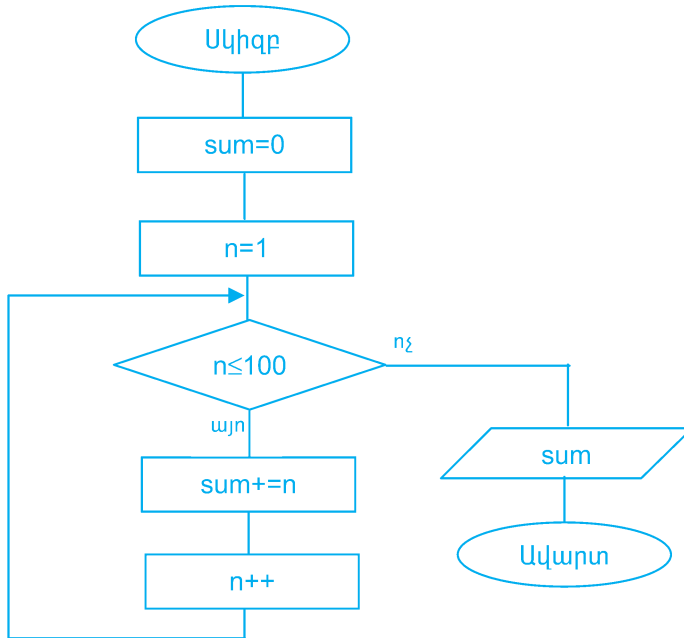
Վերը բերված գործընթացը C++ լեզվում նկարագրվում է *նախապայմանով ցիկլի օպերատորի* միջոցով՝

while (պայմանական արտահայտություն) ցիկլի մարմին:

որտեղ *պայմանական արտահայտությունը* ցանկացած տրամաբանական կամ թվաբանական արտահայտություն է, իսկ *ցիկլի մարմինը*՝ ցանկացած օպերատոր կամ բաղադրյալ օպերատոր:

Վերն ասվածից բխում է, որ եթե ցիկլի մարմնում կրկնության պայմանի արժեքը *true*-ից *false*-ի փոխող օպերատոր չներառվի՝ *անվերջ կրկնվող ցիկլ* կունենանք:

Կրկնության (ցիկլի) գործընթացին ծանոթանանք հետևյալ օրինակով՝ *գումարել 1-ից 100 միջակայքի ամբողջ թվերը*:



Նկ. 2.9. Նախապայմանով ցիկլային գործընթացի օրինակ

Բերված ալգորիթմում գումարը հաշվելու համար նախատեսված *sum* փոփոխականը ստացել է 0, իսկ հերթական գումարելիի արժեքի համար նախատեսված *n*-ը՝ 1 արժեքը: $n \leq 100$ պայմանը ճշմարիտ լինելու դեպքում կիրառործվեն հետևյալ բոկները՝ հերթական գումարելիի ավելացումը *sum*-ի մեջ՝ $sum+=n$; և հաջորդ գումարելիի ստացումը՝ $n++$: Ակնհայտ է, որ հարյուրերորդ գումարելին ($n = 100$) ավելացնելուց հետո *n*-ի մեջ ստացված թիվը (101) այլևս չի բավարարի ցիկլի կրկնման պայմանին ($n \leq 100$), ցիկլի կրկնության պայմանը կստանա *false* արժեք՝ այսպիսով ավարտելով ցիկլի գործընթացը:

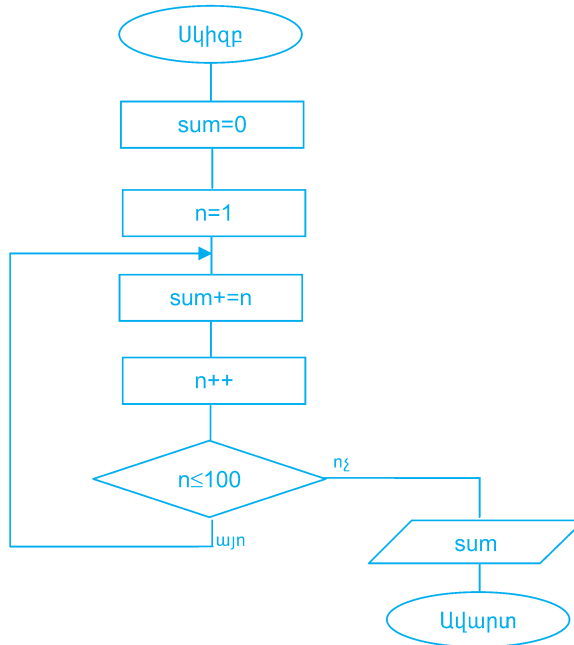
Գրենք բերված բլոկ-սխեմային համապատասխանող ծրագիրը.

```

#include <iostream.h>
void main()
{ unsigned int sum=0, n=1;
  while (n<=100)
  { sum+=n;
    n++;
  }
  cout <<sum <<endl;
}
  
```

Ակնհայտ է, որ ծրագրում ցիկլի մարմինը կազմող բաղադրյալ օպերատորը կարելի է վերակազմավորել մեկ օպերատորի՝ $sum+ = n++$:

Այժմ միևնույն խնդրի լուծման ընթացքը նկարագրենք նաև մեկ այլ սխեմայով՝



Նկ. 2.10. Հեղապայմանով ցիկլի կիրառման օրինակ

Ցիկլի կրկնման պայմանն այժմ տեղադրված է ցիկլի մարմնից (կրկնվող մասից) հետո: Նախապայմանով ցիկլի և այս գործընթացի հիմնական տարբերությունն այն է, որ այստեղ ցիկլի մարմինն **անպայման մեկ անգամ իրագործվում է**՝ նախքան ցիկլի կրկնման պայմանի ստուգումը: Եթե նախապայմանով ցիկլի մարմինը կարող է ոչ մի անգամ չիրագործվել (կրկնության գործընթացը սկսելու պահին ցիկլի կրկնման պայմանի *false* արժեք ունենալու դեպքում), ապա այժմ դա բացառվում է:

Բերված սխեման ծրագրում իրականացվում է մեկ այլ, այսպես կոչված, **հեղապայմանով ցիկլի օպերատորի** միջոցով, որն ունի հետևյալ տեսքը.

do

ցիկլի մարմին

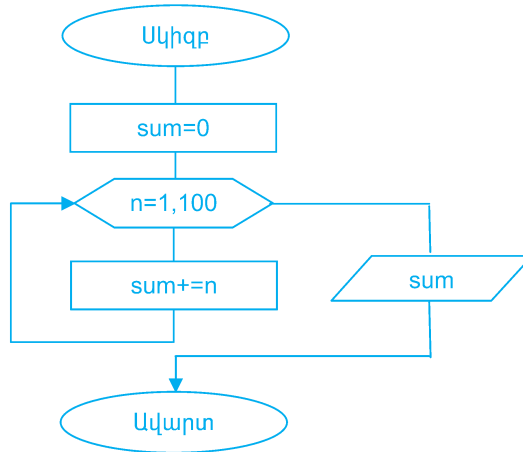
while (ցիկլի կրկնման պայման);

որտեղ *ցիկլի մարմինը* կարող է կազմված լինել մեկ կամ մի քանի օպերատորներից:

Հետագա հնարավոր սխալներից խուսափելու համար (հետապայմանով և նախապայմանով ներդրված ցիկլերի առկայության դեպքում) խորհուրդ է տրվում հետապայմանով ցիկլի օպերատորի մարմինը ձևակերպել որպես բաղադրյալ օպերատոր (առնել ձևավոր փակագծերի մեջ)՝ չնայած C++-ի կոմպիլյատորը չի պարտադրում:

Թե՛ նախապայմանով և թե՛ հետապայմանով ցիկլային գործընթացներ կազմակերպելիս (ինչպես բերված օրինակներում) կարևոր է *ցիկլի պարամետրը սկզբնարժեքավորել ցիկլի մարմնից առաջ*, այլապես *ցիկլային գործընթացի ելքը կլինի անորոշ*:

Վերը նկարագրված 1-ից հարյուր թվերի գումարումը փորձենք իրագործել մեկ այլ սխեմայով՝ ցիկլի (մոդիֆիկացիայի) բլոկի կիրառմամբ (նկ. 2.11):



Նկ. 2.11. Պարամետրով ցիկլի օրինակ

Ցիկլի կամ մոդիֆիկացիայի բլոկն աշխատանքն առավել ավտոմատ դարձնելու միջոց է, քանի որ այն մի քանի գործողություն է իրականացնում՝ ցիկլի պարամետրի սկզբնարժեքավորում ($n = 1$), պարամետրի ընթացիկ արժեքի և հնարավոր վերջին արժեքի համեմատում ($n \leq 100$), ցիկլի մարմինն իրագործելուց հետո նորից մոդիֆիկացիայի բլոկ վերադառնալիս ցիկլի պարամետրի փոփոխում (այս դեպքում՝ աճ՝ $n += 1$):

Նման բլոկ-սխեման ծրագրում իրագործվում է **պարամետրով ցիկլի օպերատորի** միջոցով, որն ունի հետևյալ տեսքը՝

for (պարամետրի նախնական արժեքի վերագրում; ցիկլի կրկնման պայման; պարամետրի փոփոխում) ցիկլի մարմին;

Այստեղ *ցիկլի պարամետրը* կարող է լինել ինչպես ամբողջթվային, այնպես էլ իրական տիպի:

Այս դեպքում, եթե *ցիկլի մարմինը* մեկից ավելի հրահանգներ է պարունակում, անհրաժեշտ է այն ձևակերպել որպես բաղադրյալ օպերատոր (առնել ձևավոր փակագծերի մեջ):

for-ին հաջորդող փակագծերում կետ-ստորակետով բաժանված երեք մասերից յուրաքանչյուրն իր հերթին կամ բոլորն իրար հետ կարող են բացակայել՝ *for (; ;)*: Եթե բացակայում է առաջին կետ-ստորակետին նախորդող մասը, ապա ենթադրվում է, որ ցիկլի պարամետրը սկզբնարժեքավորվել է ցիկլի օպերատորից առաջ: Եթե բացակայում է ցիկլի կրկնման պայմանը, ապա անհրաժեշտ է ցիկլի մարմնում կրկնման գործընթացն ընդհատող միջոց նախատեսել՝ այլապես անվերջ կրկնվող ցիկլ կառուցանա: Եթե բաց է թողնված պարամետրի արժեքը փոփոխելու մասը, ապա անհրաժեշտ է ցիկլի մարմնում պարամետրի արժեքը փոփոխող հրահանգ ներառել:

Ընդհանրապես, *for*-ի վերնագիրն ավարտող կոր փակագծին անմիջապես հաջորդող կետ-ստորակետը ցիկլի մարմինը կազմավորում է որպես դատարկ օպերատոր: Այսպիսով, ցիկլի մարմինը (որը կհաջորդեր այդ կետ-ստորակետին) չէր իրա-

գործվի և ոչ մի անգամ:

Բերենք վերը նկարագրված բլոկ-սխեմային համապատասխանող ծրագիրը.

```
#include <iostream.h>
void main()
{   unsigned int sum=0, n;
    for (n=1; n <= 100; n++)
        sum+=n;
    cout <<sum <<endl;
}
```

Նույն՝ 1-ից 100 թվերի գումարը կարելի է հաշվել նաև ցիկլի օպերատորի հետևյալ ձևակերպմամբ՝

```
for (n=100; n>=1;n--) sum+=n;
```

Այս դեպքում ասում են, որ ունենք **նվազող պարամետրով ցիկլ**:

Պարամետրով ցիկլի գրելաձևը թույլատրում է պարամետրի նախնական արժեքի տրման և դրա արժեքի փոփոխման մասերում *ստորակերպի գործողության* միջոցով կցված մի քանի օպերատորներ տալ, օրինակ՝

```
#include <iostream.h>
void main()
{   unsigned int i, sum, n;
    for (sum=0, n=1; n<=100; sum+=n, n++);
    cout <<sum;
}
```

Մենք արդեն ընտրության *switch* օպերատորի աշխատանքից ծանոթ ենք *break*-ի գործողությանը: Այս միջոցը կիրառում են նաև **կրկնողական բնույթի գործողություններն ընդհատելու համար**:

Օրինակ, եթե պարամետրով ցիկլի վերնագրային մասում բաց է բողմված ցիկլի կրկնման պայմանը, ապա կարելի է ցիկլի մարմնում ընդգրկված *break*-ի միջոցով ցիկլի ընթացքն ընդհատել՝

```
# include <iostream.h>
void main ()
{   unsigned int sum=0, n;
    for (n=1; ; n++)
        {   if (n>100) break;
            sum+=n;
        }
    cout <<sum;
}
```

break-ն ընդհատում է ցիկլի մեջ ընդգրկված գործողությունների իրագործումը և ծրագրի հետագա ընթացքը շարունակում է ցիկլի մարմնին հաջորդող (*cout <<sum;*) օպերատորը:

Ցիկլի ընթացքը մասնակիորեն ընդհատելու համար կիրառում են *continue* օպերատորը. սա ցիկլի պարամետրի ընթացիկ արժեքի համար ընդհատում է *continue*-ին հաջորդող օպերատորների իրականացումը՝ կրկին ցիկլի սկիզբ վերադառնալով: Օրինակ՝ գումարել 1-ից 100 միջակայքի 3-ին բազմապատիկ թվերը.

```
# include <iostream.h>
void main()
{   unsigned int sum=0, n;
    for (n=3; n<=99; n++)
        { if (n%3 != 0) continue;
          sum+=n;
        }
}
```

Այսպիսով, երբ $n \% 3 \neq 0$ (այսինքն՝ n -ը բազմապատիկ չէ 3-ին), իրագործվում է *continue* հրամանը, որը, բաց թողնելով ցիկլի մարմնում իրեն հաջորդող $sum+=n$; հրամանը, գործողությունները շարունակում է իրագործել ցիկլի սկզբից (n -ը աճեցվում է 1-ով և այլն), հակառակ դեպքում *continue*-ն չի իրագործվում. կատարվում է $sum+=n$; օպերատորը: Ասենք, որ այս օրինակը բացառապես բերված է *continue*-ի աշխատանքը պարզաբանելու նպատակով, այլպես 3-ին բազմապատիկ թվերի գումարը կարելի է առավել ռացիոնալ կերպով հաշվել՝

```
for (n=3; n<=99; n+=3) sum+=n;
```

ցիկլի միջոցով:

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ **Խորհուրդ է փոխում ցիկլի կրկնությունների քանակը որոշող պարամետրը բնութագրել ամբողջ փիպի (սահող ստորակետով թվերը համակարգչում որոշ մոդավորությամբ են ներկայացվում, որը կարող է ցիկլի ավարտի պայմանը ստուգելիս սխալ արդյունքի հանգեցնել):**
- ◆ **for-ի վերնագրային մասում կեր-ստորակետերի փոխարեն ստորակետ կիրառելը քերականական սխալ է:**
- ◆ **for-ի մարմնում continue-ի կիրառումը հանգեցնում է հեղազա գործողությունների կանխմանը, ցիկլի կրկնման պարամետրերի փոփոխմանը և սպա՝ կրկնության պայմանի ստուգմանը:**
- ◆ **for և while ցիկլերի օպերատորների աշխատանքները հիմնականում համարժեք են՝ բացառությամբ այն դեպքի, երբ while նախապայմանով ցիկլի մարմնում կիրառված continue-ն փակակայքով է ցիկլի պարամետրը փոփոխող օպերատորից առաջ. այս դեպքում անվերջ կրկնվող ցիկլ ունենալու վտանգ է առաջանում:**



1. *While, do...while* և *for* օպերատորների կիրառմամբ հաշվել [50;100] միջակայքի զույգ արժեք ունեցող քվերի զուամարը:
2. Գտնել *ցիկլի այն օպերատորները, որոնց գրառման մեջ սխալ է բույլ տրված*.
 $s = 0; \text{ for } (i = 0, i < 10, i++ = 1) s++ = i;$
 $k = 5; \text{ while } (k > 0); k--;$
 $c = 0; \text{ do } \{y++ = ++c;\} \text{ while } (c < 5);$
 $l = 100; \text{ for } (k = 5; k < 15; k++ = 2); l-- = k;$
3. *for* օպերատորի միջոցով հաշվել տրված n բնական քվի ֆակտորիալը ($n!$), որպես $n! = 1 \cdot 2 \cdot 3 \cdot \dots \cdot n$:
4. Կազմել հավելված 3-ի այս բեմային առնչվող խնդիրների լուծման բլոկ-սխեմաներն ու ծրագրերը:

§ 2.7 ՄԻԱԶԱՓ ԶԱՆԳՎԱԾՆԵՐ

Զանգվածը միևնույն տիպի ունեցող համանուն տարրերի հավաքածու է, որտեղ յուրաքանչյուր տարր բնորոշվում է իր **հերթական համարով** (**ինդեքսով**):

Զանգվածը համակարգչի հիշողությունում հաջորդական անընդմեջ տիրույթ է զբաղեցնում:

Զանգվածի որևէ տարրին դիմելու համար անհրաժեշտ է նշել ոչ միայն զանգվածի անունը, այլև տվյալ տարրի **դիրքի համարը** (**ինդեքսը**), ընդ որում՝ զանգվածի տարրերի համարակալումը սկսվում է 0-ից: Օրինակ, 10 իրական տարրեր պարունակող a զանգվածը, որը հայտարարվում է

$$\text{double } a[10];$$

տեսքով, որտեղ $[]$ -ում ներառված 10 թիվը ցույց է տալիս, որ տվյալ դեպքում a զանգվածը կարող է ամենաշատը 10 տարր պարունակել: Այս զանգվածը կազմված է $a[0]$, $a[1]$, ..., $a[9]$ անուններով տարրերից, որտեղ $[]$ փակագծերում նշվել են տարրերի հերթական **համարները՝ ինդեքսները**: Օրինակ, a զանգվածի 4-րդ համարը կրող (զանգվածի 5-րդ տարր) արժեքը 4.21 դարձնելու համար պետք է կատարել հետևյալ վերագրումը՝ $a[4]=4.21;$:

Եթե զանգվածի տարրը որոշվում է մեկ ինդեքսի միջոցով, ապա այդպիսի զանգվածն անվանում են **միաչափ**: Վերը բերված օրինակում a զանգվածը միաչափ է:

Զանգվածի տարրերը կարելի է սկզբնարժեքավորել զանգվածը հայտարարելիս:

Ընդ որում՝ եթե սկզբնարժեքավորելիս տրվող *արժեքների քանակը մեծ է* զանգվածի հայտարարման պահին տրված տարրերի քանակից՝ քերականական սխալ է առաջանում: Իսկ եթե սկզբնարժեքավորմամբ տրվող *արժեքների քանակը* հայտարարվող զանգվածի տարրերի քանակից *պակաս է*, ապա մնացած «ավել» տարրերը սկզբնարժեքավորվում են 0 արժեքներով: Օրինակ, `int a[5] = {0};`; հայտարարությամբ նախ `a[0]`-ն ստանում է 0 արժեք, որից հետո, ըստ վերը բերված կանոնի, 0 արժեքներ են ստանում նաև զանգվածի մնացած տարրերը: Եթե հայտարարման ժամանակ զանգվածի տարրերի քանակը (հետագայում՝ չափ) չի տրվում, ապա այն ավտոմատ հավասարեցվում է սկզբնարժեքավորման ցուցակում ներառված արժեքների քանակին: Օրինակ՝

```
int a[] = {5, -7, 8};
```

հայտարարությամբ *a*-ն կահանանվի որպես ամենաշատը 3 տարր պարունակող զանգված:

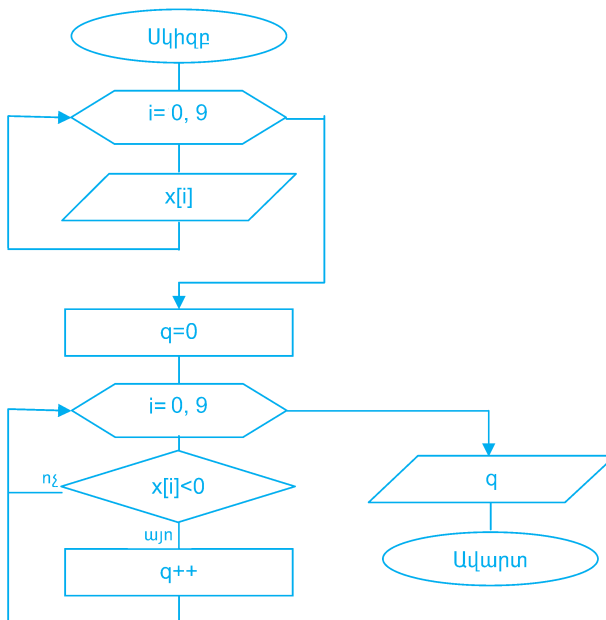
Եթե զանգվածի տարրերի քանակը ծրագրի կատարման ընթացքում մնալու է անփոփոխ, ապա այն կարելի է հայտարարել *const* նույնարկիչի տակ, օրինակ, հետևյալ կերպ.

```
const int k=10; int a[k];
```

Մա նշանակում է, որ *k*-ն ծրագրի կատարման ամբողջ ընթացքում այլևս չի կարող արժեքը փոփոխել՝ հաստատուն է. ընդ որում, այսպես կոչված, **անունակիր հասարակություն** է, որը նախատեսված է միայն ընթերցման համար:

Խնդիր. *Չաշվել 10 իրական տարրեր պարունակող զանգվածի բացասական տարրերի քանակը:*

Կազմենք խնդրի լուծման բլոկ-սխեման (նկ. 2.12):



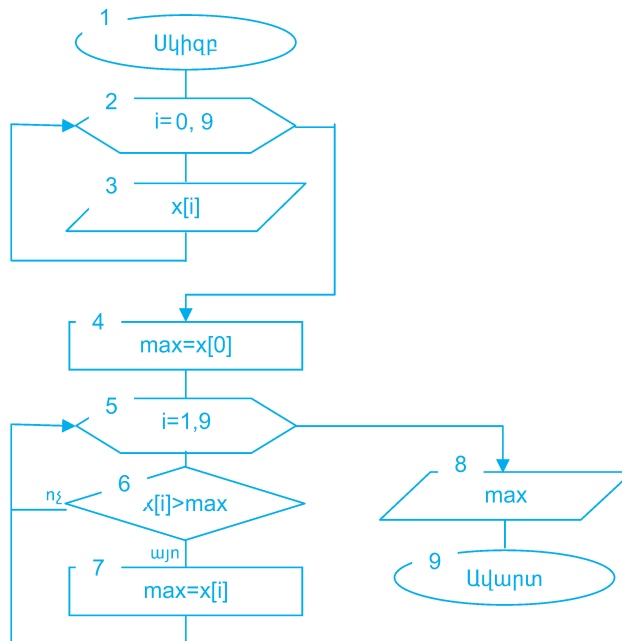
Նկ. 2.12. *Չանգվածի բացասական տարրերի քանակը հաշվելու ալգորիթմ*

Բլոկ-սխեման սկսվել է պարամետրով ցիկլի միջոցով x զանգվածի տարրերի ներմուծմամբ: Այնուհետև պահանջվող քանակը հաշվելու համար նախատեսված q փոփոխականին վերագրվել է նախնական 0 արժեք և $i=0, 9$ ցիկլի միջոցով հաշվարկվել է պահանջվող տարրերի քանակը: Գրենք համապատասխան ծրագիրը.

```
#include <iostream.h>
void main ()
{   double x[10]; int i,q;
    for (i=0; i<10; i++)
        cin >> x[i];
    q=0;
    for (i=0; i<10; i++)
        if (x[i]<0) q++;
    cout << q << endl;
}
```

Միաշափ զանգվածների վերաբերյալ ևս մեկ խնդիր լուծենք.

Խ ն դ ի թ. Որոշել տրված 10 իրական տարրեր պարունակող միաշափ զանգվածի մեծագույն տարրի արժեքը:



Նկ. 2.13. Միաշափ զանգվածի մեծագույն տարրը որոշելու ալգորիթմ

Չանգվածի տարրերի արժեքների ներմուծումից հետո 4-րդ բլոկով կատարվել է $max=x[0]$ վերագրումը: Այսպիսով, $x[0]$ տարրը ենթադրաբար համարվել է մեծագույն, իսկ հետագայում $i=1, 9$ ցիկլի միջոցով այն համեմատվել է հաջորդ տարրերի հետ, և եթե ավելի մեծ տարր է հայտնաբերվել, ապա max փոփոխականի արժեքը փոխարին-

վել է դրանով: Ցիկլի ավարտին max -ը կպարունակի զանգվածի մեծագույն տարրի արժեքը, որն էլ կարտածվի δ -րդ բլոկով: Կազմենք համապատասխան ծրագիրը.

```
#include <iostream.h>
void main ( )
{ int i, x[10], max;
  for (i=0; i<=9; i++)
    cin >> x[i];
  max=x[0];
  for (i=1; i<=9; i++)
    if (x[i]>max) max=x[i];
  cout << "max=" << max;
}
```

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ **Քառակուսի [] փակագծերն իրականում ինդեքսավորման գործողություն են և գործողությունների կատարման նույն առաջնահերթությունն ունեն, ինչ կոր () փակագծերը:**



1. Ի՞նչ է զանգվածը:
2. Ո՞ր զանգվածն են անվանում միաչափ:
3. Կարո՞ղ է զանգվածի տարրի ինդեքսն իրական թիվ լինել:
4. Կազմել հավելված 3-ի այս թեմային առնչվող խնդիրների լուծման բլոկ-սխեմաներն ու ծրագրերը:

§ 2.8 ԵՐԿՉԱՓ ՁԱՆԳՎԱԾՆԵՐ

$C++$ -ում զանգվածները կարող են մինչև **12** հատ ինդեքսներ ունենալ. ուսումնասիրենք **երկչափ** (երկու ինդեքս պարունակող) **զանգվածները**:

Երկչափ զանգվածը ակնառու պատկերացնելու համար դիտենք դասարանի նստարանների շարքերը: Ենթադրենք, դրանք դասավորված են **4** շարքով, իսկ յուրաքանչյուր շարքում կա **3**-ական սեղան. համարակալենք սեղաններն այնպես, որ այդ համարներով միարժեքորեն որոշվի սեղանի գտնվելու շարքն ու շարքում դրա դիրքը.

$S[1][1]$	$S[1][2]$	$S[1][3]$	$S[1][4]$
$S[2][1]$	$S[2][2]$	$S[2][3]$	$S[2][4]$
$S[3][1]$	$S[3][2]$	$S[3][3]$	$S[3][4]$

Ինչպես տեսնում եք, շարքը բնորոշող համարը քառակուսի փակագծերի մեջ առված թվերից երկրորդն է, իսկ առաջինը՝ տվյալ շարքում նստարանի հերթական հա-

մարը: Նման եղանակով կարգավորված տվյալների համախումբն անվանում են **երկ-չափ զանգված**: Երկչափ զանգվածի ցանկացած տարրին դիմելու համար անհրա-ժեշտ է երկու ինդեքս կիրառել. **առաջին ինդեքսը** համարում են տարրի գտնվելու **դո-դի**, իսկ **երկրորդը` սյան համարը**: Նույն սկզբունքով կարելի է սահմանել նաև բազ-մաչափ զանգվածները:

C++-ում տողն ու սյունը սկսում են համարակալել 0-ից, այսինքն` վերը բերված օրինակն այստեղ կընդունի հետևյալ տեսքը.

```
S[0][0]    S[0][1]    S[0][2]    S[0][3]
S[1][0]    S[1][1]    S[1][2]    S[1][3]
S[2][0]    S[2][1]    S[2][2]    S[2][3]
```

Երկչափ զանգված հայտարարելիս պետք է նշել ինչպես տողերի, այնպես էլ սյու-նների առավելագույն քանակը, օրինակ`

```
int x[10][20];
```

որտեղ 10-ը ցույց է տալիս տողերի, իսկ 20-ը` սյունների առավելագույն քանակը:

Երկչափ զանգվածի տարրերը նույնպես կարելի է սկզբնարժեքավորել զանգվածը հայտարարելիս, օրինակ`

```
int a[2][2] = {{3,1}, {-4,2}};
```

որի դեպքում կունենանք հետևյալ երկչափ զանգվածը.

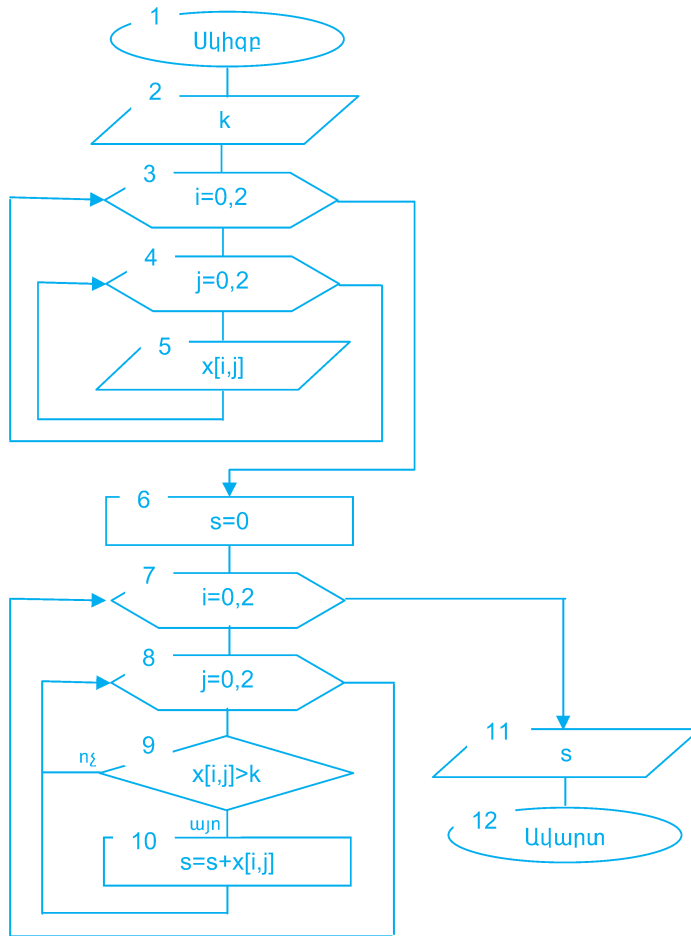
	0 սյուն	1 սյուն
0 դող	3	1
1 դող	-4	2

Այստեղ յուրաքանչյուր տող սկզբնարժեքավորելու համար գործում են նույն կա-նոնները, ինչ միաչափ զանգվածը սկզբնարժեքավորելիս:

Երկչափ զանգվածների հետ կապված աշխատանքին ավելի մոտիկից ծանոթա-նալու նպատակով մի քանի խնդիր լուծենք:

Խ ն դ ի ր . *Տրված է ամբողջ տիպի 3x3 (3 տող և 3 սյուն) տարրեր պարունա-կող երկչափ զանգված: Հաշվել տրված k ամբողջ թվից մեծ արժեք ունեցող տարրերի գումարը:*

Ինչպես երևում է բլոկ-սխեմայից (նկ.2.14)` երկչափ զանգված ներմուծելու համար մեկը {4} մյուսի {3} մեջ ներդրված ցիկլեր են կիրառվել: **Ներդրված ցիկլերն** աշխա-տում են հետևյալ կերպ. նախ արտաքին {3} ցիկլի i պարամետրը ստանում է իր սկզբնական` 0 արժեքը, և ղեկավարումը տրվում է ներդրված {4} ցիկլին: Վերջինս ցիկլի սովորական, մեզ արդեն հայտնի սխեմայով է աշխատում, այսինքն` j-ն փո-փոխվելով 0-ից 2` ներմուծվում են i-րդ (այս պահին` 0-րդ) տողի տարրերը, այնու-հետև ղեկավարումը կրկին տրվում է {3} բլոկին, որտեղ i-ն աճելով ստանում է 1 ար-ժեքը, և ամեն ինչ ընթանում է այնպես, ինչպես i=0 արժեքի դեպքում, այսինքն` այժմ ներմուծվում են 1 համարով տողի տարրերը: Նույնը կատարվում է նաև i=2-ի դեպ-քում: Աշխատանքի այսպիսի ընթացքը հատկանշական է ցանկացած ներդրված ցիկ-լերի համար:



Նկ. 2.14. Երկչափ զանգվածի փարթերի գումարը հաշվելու ալգորիթմ

Այժմ կազմենք համապատասխան ծրագիրը.

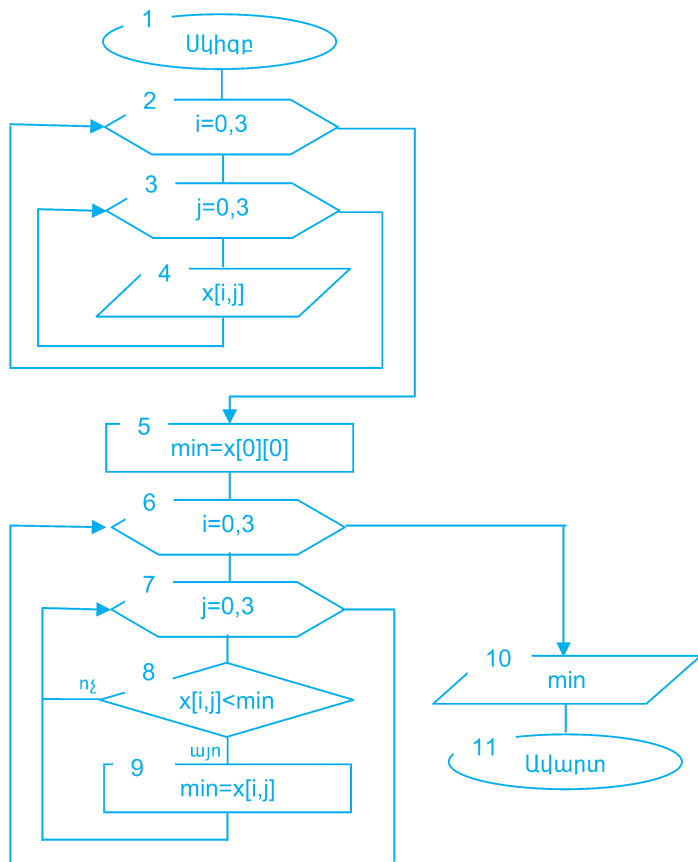
```

#include <iostream.h>
const int n=3;
void main ()
{ int x[n][n], i, j, k, s=0;
  cout << "k="; cin >> k;
  for (i=0; i<n; i++) //1
  for (j=0; j<n; j++) //2
  cin >> x[i][j]; //3
  for (i=0; i<n; i++)
  for (j=0; j<n; j++)
  if (x[i][j]>k) s+=x[i][j];
  cout << s << endl;
}
  
```

Ծրագրում //1 և //2 մեկնաբանությամբ ցիկլերի միջև ձևավոր փակագծերի անհրաժեշտություն չկա, քանի որ i ցիկլում մեկ օպերատոր կա //2-ը, իսկ //3-ը //2-ի միակ կրկնվող օպերատորն է:

Կազմենք հետևյալ խնդրի լուծման բլոկ-սխեման ու ծրագիրը ևս. *տրված է 4x4 իրական տարրեր պարունակող երկչափ զանգված: Հաշվել և արտածել զանգվածի փոքրագույն տարրի արժեքը:*

Նախ կազմենք խնդրի լուծման բլոկ-սխեման (նկ. 2.15):



Նկ. 2.15. Երկչափ զանգվածի փոքրագույն արտքը որոշելու ալգորիթմ

Այս ալգորիթմը նման է միաչափ զանգվածի մեծագույն տարրը որոշելու ալգորիթմին. նախ *min* փոփոխականի մեջ պահվել է զանգվածի տարրերից առաջինը (ընդ որում, կարևոր չէ, թե զանգվածի որ տարրի արժեքն այս պահին կհամարվի փոքրագույն), որից հետո 6-րդ և 7-րդ բլոկներով կազմավորված ներդրված ցիկլերի միջոցով ենթադրյալ փոքրագույնի (*min*) հետ համեմատվել են զանգվածի բոլոր մնացած տարրերն ու արժեքով առավել փոքր տարրը 9-րդ բլոկում վերագրվել է *min*-ին: Ներդրված 6-րդ և 7-րդ ցիկլերի աշխատանքի ավարտին *min*-ի մեջ գրված կլինի զանգվածի փոքրագույն տարրը, որի արժեքն արտածվել է 10-րդ բլոկով:

Կազմենք ծրագիրը.

```
#include <iostream.h>
const int n=4;
{   double x[n][n], min;
    int i,j;
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            cin >> x[i][j];
    min=x[0][0];
    for (i=0; i<n; i++)
        for (j=0; j<n; j++)
            if (x[i][j]<min) min=x[i][j];
    cout << "min=" <<min;
}
```

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ *m* փող *և* *n* սյուն պարունակող զանգվածներն անվանում են *m*-ը *n*-ի վրա, կամ *m* × *n* փարս պարունակող զանգված:
- ◆ Երկչափ *n* × *n* փարս պարունակող զանգվածի վերին շախ (0, 0 համարով) փարսից մինչև ստորին աջ (*n* − 1, *n* − 1 համարով) փարսն ընկած անկյունագիծն անվանում են գլխավոր անկյունագիծ: Գլխավորի վրա ընկած փարսերի համար սյան (*j*) և փողի (*i*) համարները (ինդեքսները) իրար հավասար են (*j* = *i*):
- ◆ Երկչափ *n* × *n* փարս պարունակող զանգվածի վերին աջ (0, *n* − 1 համարով) փարսից մինչև ստորին շախ (*n* − 1, 0 համարով) փարսն ընկած անկյունագիծն անվանում են օժանդակ անկյունագիծ: Օժանդակի վրա ընկած փարսերի համար բնորոշ է փողի (*i*) և սյան (*j*) համարների հեղեղյալ կապը՝ *i* + *j* = *n* − 1:



1. Երկչափ զանգվածի առաջին փողի փարսերը կարո՞ղ են լինել սիմվոլային փիպի, իսկ մնացած փողերինը, օրինակ՝ ամբողջ փիպի:
2. Եթե զանգվածի փարսը բնորոշվում է որպես $x[a][b][c]$, ապա զանգվածը
 - ա) երկչափ է,
 - բ) միաչափ է,
 - գ) եռաչափ է,
 - դ) $a * b * c$ չափի է:
3. Կազմել հավելված 3-ի այս թեմային առնչվող խնդիրների լուծման բլոկ-սխեմաներն ու ծրագրերը:

§ 2.9 ՀՂՈՒՄՆԵՐ: ՑՈՒՑԻՉՆԵՐ

Հղումը իդենտիֆիկատորի երկրորդ անվանումն է այնպես, ինչպես, օրինակ, Հովիկը Հովհաննես անունով անձի համար:

Հղումը մի միջոց է, որը հնարավորություն է տալիս իդենտիֆիկատորին հարկացված հիշողությունն անվանել տվյալ իդենտիֆիկատորից տարբեր մեկ այլ անվամբ ևս:

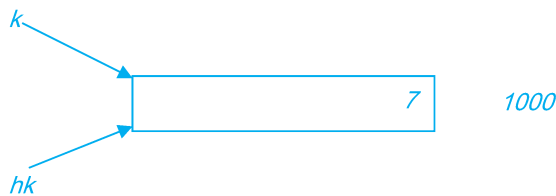
Հղումը հայտարարում են հետևյալ կերպ.

տիպ & իդենտիֆիկատոր 2 = իդենտիֆիկատոր 1;
հղում

որտեղ *իդենտիֆիկատոր 1*-ը պետք է նախօրոք հայտարարված լինի և տիպով համընկնի հայտարարվող հղման տիպի հետ: Օրինակ՝

`int k=7; int &hk=k;`

Այս հայտարարությունից հետո *hk*-ն հանդիսանում է *k*-ի հղումը: Այսպիսով, եթե *k* ամբողջ տիպի փոփոխականի համար հատկացվել էր հիշողության 1000 հասցեն, ապա այս հայտարարումից հետո նույն հասցեն կունենա ևս մեկ՝ *hk* անվանումը (նկ. 2.16)



Նկ. 2.16. Հղման օրինակ

Այժմ `cout << hk;` և `cout << k` հրամանների արդյունքում էկրանին կարտածվի միևնույն 7 թիվը:

Սեկ անգամ հայտարարված հղումը միշտ «հավատարիմ» է իր առաջին և միակ «հոմանիշին»: Սակայն նույն իդենտիֆիկատորը կարող է բազմաթիվ հղումներ՝ երկրորդ անուններ ունենալ՝

`int &d = k;`

`int &p = k;` և այլն:

Հետևելով ծրագրի հետևյալ հատվածի աշխատանքին՝

```
.....
int c=-7;
int & hc=c;
cout << "c=" << c << endl ;
cout << "hc=" << hc << endl ;
hc+=5 ;
cout << "c=" << c <<endl ;
cout << "hc=" << hc << endl ;
```

կտեսնենք հետևյալը՝

```
c=-7
hc=-7
c=-2
hc=-2:
```

Այսպիսով, c փոփոխականի հղման՝ hc -ի արժեքի $hc+ = 5$ փոփոխությունը նույն ձևով «անդրադառնում է» c փոփոխականի արժեքի վրա:

Այժմ ուսումնասիրենք C++-ի մի շատ կարևոր հասկացության՝ **ցուցիչների** հետ կապված աշխատանքը:

Ցուցիչը հապուկ ձևով հայտարարված փոփոխական է, որի համար որպես արժեք ծառայում է հիշողության հասցեն:

Ասում են, որ փոփոխականն իրեն հատկացված հիշողությունում պահված արժեքին դիմում է *ուղղակի* (անվան միջոցով), մինչդեռ նույն *փոփոխականի ցուցիչը՝ անուղղակիորեն* (հասցեի միջոցով): Արժեքին ցուցի միջոցով դիմելու եղանակն անվանում են **անուղղակի հասցեավորում**:

Ցուցիչները հայտարարում են հետևյալ կերպ՝

*տիպ * իդենտիֆիկատոր ;*

Օրինակ՝ *double *pt ;*
*double * k ;*

Ինչպես տեսնում եք՝ հայտարարման մեջ $*$ -ը կարող է անմիջապես կից լինել ինչպես իդենտիֆիկատորին, այնպես էլ տիպը բնորոշող բառին՝ երկու հայտարարություններն էլ ցուցի ճիշտ և համարժեք հայտարարություններ են:

Եթե բերված է, օրինակ, *int *p, k*; հայտարարությունը, ապա հայտարարված է p ցուցիչ, որն «ունակ է ցույց տալու» ցանկացած ամբողջ տիպի պարամետրի վրա, մինչդեռ k -ն ուղղակի *int* տիպի փոփոխական է. որպեսզի k -ն նույնպես ըստ հայտարարման հանդիսանա *int* տիպի ցուցիչ, ապա հայտարարությունը պետք է ունենա, օրինակ, *int *p, *k*; տեսքը:

Բացի իր տիպն ունեցող փոփոխականների հասցեներից, ցուցիչը չի կարող այլ տիպի փոփոխականի հասցե պարունակել:

Ցուցիչները կարելի է սկզբնարժեքավորել ինչպես դրանք հայտարարելիս, այնպես էլ հետագայում:

Հասցեից բացի ցուցիչը կարող է ունենալ միայն մեկ արժեք՝ **0** կամ, որ նույնն է՝ **NULL**:

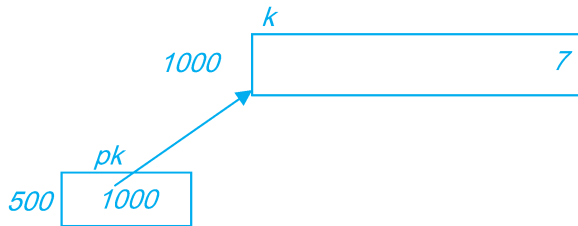
0 կամ NULL արժեք ունեցող ցուցիչը համարվում է ազատ, որևէ փոփոխականի վրա ցույց չգրվող:

Ցուցիչը արժեքավորում են **հասցե բացահայտելու & գործողության** միջոցով՝ օրինակ՝ հետևյալ կերպ.

```
int k=7 ;
```

```
int *pk = &k;
```

Այս վերագրումից հետո *pk* ցուցիչը կպարունակի *k* փոփոխականի հասցեն (նկ. 2.17):



Նկ. 2.17. Ցուցի օրինակ

Ըստ բերված օրինակի՝ *pk* ցուցիչը, որին հիշողության 500 հասցեն է տրամադրված, պարունակում է 1000-ը, որը *k*-ի հասցեն է: Ասում են, որ *pk*-ն ցույց է տալիս *k* փոփոխականի վրա:

*-ն անվանում են **անուղղակի հասցեավորման գործողություն**, որի միջոցով հնարավորություն ենք ստանում դիմել ցուցչի ցույց տված հասցեի պարունակությանը (արժեքին):

Այսպիսով, վերը բերված օրինակում `cout << *pk`; հրամանի արդյունքում էկրանին կհայտնվի *k* փոփոխականի արժեքը՝ 7, իսկ `*pk = -10`; վերագրումից հետո *k* փոփոխականի արժեքը կհավասարվի -10-ի, քանի որ *pk*-ի ցույց տված *k* փոփոխականին տրամադրված հասցեում (1000) կգրվի -10 արժեքը:

Ցուցիչների աշխատանքին ծանոթանալու նպատակով բերենք հետևյալ ծրագիրը.

```
#include <iostream.h>
void main()
{ int a, b, *pa = NULL, *pb = NULL;
  a=5; b=7;
  pa=&a; pb=&b; //1
  cout << "a+b=" << a+b << endl; //2
  // hima nuyngumar@stanang cucichnerov
```

```

cout << "a+b=" << *pa + *pb << endl;           //3
cin >> a >> b;
cout << "a-b=" << a-b << endl;               //4
// hima myn tarberutyun@ stanang cucichnerov
cout << "a-b=" << *pa - *pb << endl;         //5
}

```

Ծրագրում հայտարարվել են a և b ամբողջ տիպի փոփոխականները և նույն տիպի երկու ցուցիչներ՝ $*pa$ և $*pb$: Քանի որ ցուցիչները հայտարարման պահին չեն սկզբնարժեքավորվել, այդ պատճառով դրանց տրվել են նախնական $NULL$ արժեքներ: Այնուհետև //1 տողում թե՛ $*pa$ -ն և թե՛ $*pb$ -ն արժեքավորվել են. **հասցեի վերահսկման & գործողության** միջոցով $*pa$ -ն ստացել է a -ի, իսկ $*pb$ -ն՝ b -ի հասցեները: //2 տողում արտածման արդյունքը կլինի 12 թիվը. նույն արդյունքը կստացվի նաև //3-ի արդյունքում, քանի որ $*pa + *pb$ գումարը հաշվելիս համակարգիչը pa ցուցիչ ցույց տված հասցեի պարունակությանը (5) կավելացնի pb -ի ցույց տված հասցեի պարունակությունը (7): Մինչև նույն պատճառով //4-րդ և //5-րդ տողերի կատարման արդյունքում կստացվի մինչև նույն՝ -2 պատասխանը:

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ **Հղումը հայտարարման պահին պարզադիր պեղք է սկզբնարժեքավորվի:**
- ◆ **Չնայած ցուցիչը NULL և 0 արժեքներով սկզբնարժեքավորելը համարժեք գործողություններ են, սակայն C++-ում 0-ով սկզբնարժեքավորելը նախընտրելի է:**
- ◆ **Եթե ցուցիչը սկզբնարժեքավորվում է 0-ով, ապա այն չեափոխվում է համապատասխան փիպի ցուցիչ:**
- ◆ **0-ն միակ ամբողջ արժեքն է, որը կարելի է ցուցիչն վերագրել առանց այն նահապես ցուցիչ փիպի չեափոխելու:**



1. **Ի՞նչ է հղումը:**
2. **Կարո՞ղ է նույն հղումը փարբեր փոփոխականների երկրորդ անուն հանդիսանալ:**
3. **Քանի՞ հղում կարող է ունենալ, օրինակ, int a; հայտարարմամբ փրված a փոփոխականը:**
4. **Ի՞նչ է ցուցիչը և դրա համար ի՞նչը կարող է արժեք լինել:**
5. **Ինչպե՞ս կարելի է սկզբնարժեքավորել ցուցիչը:**
6. **Ցուցիչը հասցեից բացի ուրիշ ի՞նչ արժեք կարող է պարունակել:**
7. **Ի փարբերություն ուղիղ հասցեավորման, ինչպե՞ս է կոչվում ցուցիչ միջոցով կապարված հասցեավորումը:**

§ 2.10 ՉԱՆԳՎԱԾՆԵՐԻ ՀԵՏ ԿԱՊՎԱԾ ՅՈՒՑԻՉՆԵՐ

C++ լեզվում զանգվածները կազմակերպված են այնպես, որպեսզի զանգվածի անունն ավտոմատ հանդիսանա զանգվածի առաջին՝ 0-ական համարով տարրի ցուցիչը:

Օրինակ, եթե հայտարարված `int a[10]` զանգվածը տեղաբաշխված է հիշողության 1000 հասցեից սկսած, ապա `cout << a;` հրամանի արդյունքում էկրանին կտեսնենք 1000 թիվը՝ `a` զանգվածի 0-րդ տարրի հասցեն: Ընդ որում՝ եթե ունենք, օրինակ, `int * pa;` ցուցիչը, ապա վերագրման հետևյալ օպերատորները համարժեք են՝

ա) `pa = a;`
բ) `pa = &a[0];`

Պատճառն այն է, որ `pa = a;` հրամանով `pa` ցուցիչն վերագրվել է `a`-ի արժեքը, որը `a[0]` տարրի հասցեն է, իսկ `pa = &a[0];` հրամանով այդ նույն գործողությունն իրականացվել է առավել «բացահայտ» արտահայտությամբ:

Չանգվածի անունը, լինելով զանգվածի 0-ական համարով տարրի ցուցիչը, այլևս չի կարող այլ արժեք ընդունել՝ այն **հասարայուն ցուցիչ** է և միշտ 0 համարով տարրի հասցեն է պարունակում:

Այն ցուցիչները, որոնք ըստ հայտարարման միշտ միևնույն հասցեն են պարունակում (միևնույն *օբյեկտի* վրա են ցույց տալիս), կոչվում են **հասարայուն ցուցիչներ**: Օրինակ, եթե `int a,*p = &a;` հայտարարմամբ `p` ցուցիչը ցույց է տալիս ոչ հաստատուն արժեք ունեցող տվյալի (`a`) վրա, ապա, օրինակ, `int d[10];` և `int k,*const p1 = &k;` հայտարարություններով թե՛ `d` և թե՛ `p1` ցուցիչները հաստատուն ցուցիչներ են, որոնք կարող են ոչ հաստատուն արժեք ունեցող օբյեկտների վրա ցույց տալ՝ `d[0]`-ն ցանկացած (ոչ հաստատուն) ամբողջ տիպի արժեք կարող է պարունակել, իսկ `p1`-ն այսուհետև կհանդիսանա միայն `k`-ի ցուցիչը և այլևս չի կարող որևէ այլ օբյեկտի վրա ցույց տալ (այլ օբյեկտի հասցե պարունակել):

Ըստ հայտարարման՝ հնարավոր են ցուցիչներ, որոնք լինելով ոչ **հասարայուն ցույց են տալիս հասարայուն արժեքի կրիչ օբյեկտների վրա** (որոնք ծրագրի ընթացքում չեն կարող փոփոխվել). դրանք հայտարարվում են հետևյալ կերպ.

*const տիպ *իդենտիֆիկատոր;*

Օրինակ՝ `const double *p;` հայտարարմամբ `p`-ն «ազատ» ցուցիչ է, որը թեպետ ծրագրի կատարման ընթացքում ըստ անհրաժեշտության կարող է `double` տիպի տարբեր փոփոխականների վրա ցույց տալ, սակայն `p`-ի ցույց տված օբյեկտների արժեքները պետք է լինեն հաստատուն:

Բացի բերված դեպքերից, հնարավոր են նաև **հասարայուն ցուցիչներ**, որոնք ցույց են տալիս **հասարայուն մեծությունների վրա**. դրանք հայտարարվում են հետևյալ կերպ՝

*const տիպ *const իդենտիֆիկատոր;*

Օրինակ՝ `const int *const p = &k` հայտարարությամբ `p` ցուցիչը ցույց է տալիս `k`-ի վրա (և ոչ մի այլ օբյեկտի ցուցիչ չի կարող լինել), իսկ `k`-ի արժեքն էլ չի կարող փոփոխվել:

Ցուցիչների թվաբանություն

Չանգվածների հետ կապված՝ ցուցիչները կարող են կիրառվել թվաբանական արտահայտությունների մեջ, վերագրման և համեմատման գործողություններում: Սակայն այստեղ կիրառելի հնարավոր գործողությունները սահմանափակ են՝ ցուցիչը կարելի է ենթարկել ինկրեմենտի (++) , դեկրեմենտի (- -), դրան ամբողջ թիվ գումարելու հանել, կարելի է նաև միևնույն զանգվածի հետ կապված մի ցուցիչից հանել մյուսը:

Հիշեցնենք, որ զանգվածները համակարգչի հիշողությունում հաջորդական, իրար կից հասցեներ են զբաղեցնում:

Ենթադրենք, համակարգչում ամբողջ (*int*) թիվը 2 բայթ ծավալով հիշողություն է զբաղեցնում: Ենթադրենք նաև, որ *int a[10]*; զանգվածի 0 համարով տարրին տրվել է 1000-րդ հասցեն (նկ. 2.18).

1000	1002	1004	1006	1008	1010	1012	1014	1016	1018
<i>a</i> [0]	<i>a</i> [1]	<i>a</i> [2]	<i>a</i> [3]	<i>a</i> [4]	<i>a</i> [5]	<i>a</i> [6]	<i>a</i> [7]	<i>a</i> [8]	<i>a</i> [9]

Նկ. 2.18. Ցուցիչների թվաբանության օրինակ

Եթե սահմանենք *int *p = &a[0]*; ապա *p*-ն ցույց կտա *a*[0]-ի վրա, այսինքն՝ կայարունակի 1000 հասցեն: Թվում է, որ *p++=3*; վերագրման արդյունքում պետք է ստացվի *1000+3=1003* թիվը, սակայն քանի որ *p*-ն ոչ թե սովորական փոփոխական է, այլ ցուցիչ, որը կապված է *a* զանգվածի հետ, այն աշխատում է այլ կերպ՝ *նախքան գումարման գործողությունն իրականացնելը գումարելին բազմապատկելուն է այնքանով, որքան բայթ պարունակում է փյվյալ ցուցչի փիպն ունեցող մեծությունը*: Քանի որ *int* տիպը պարունակում է 2 բայթ, ապա նախ հաշվվում է *2*3 = 6* արժեքը և նոր միայն ստացվածն ավելացվում *p*-ի արժեքին. այսպիսով՝ *p + = 3*; հրամանի արդյունքում *p*-ն ցույց կտա *1000 + 6 = 1006* հասցեի վրա, որտեղ զանգվածի *a*[3] տարրն է պահված:

Այսպիսով, եթե *p*-ն ցուցիչ է զանգվածի առաջին՝ 0-ական համարով տարրի վրա, ապա զանգվածի տիպից անկախ՝ *p + = k*; օպերատորի արդյունքում *p*-ն ցույց կտա զանգվածի *k* համարով տարրի վրա, իսկ եթե ցույց է տալիս զանգվածի *k*-համարով տարրի վրա, ապա *p - = k*; վերագրումից հետո այն կրկին ցույց կտա 0-ական համարով տարրի վրա: Այժմ պարզ է, որ ++*p*; կամ *p++*; գործողությամբ *p*-ն ցույց կտա ընթացիկին հաջորդող, իսկ --*p*; կամ *p--*; գործողությամբ՝ ընթացիկին նախորդող տարրի վրա:

Միևնույն զանգվածի վրա ցույց տվող ցուցիչները կարելի է **իրարից հանել**. եթե *p*-ն զանգվածի 6-րդ համարով տարրն է ցույց տալիս, իսկ *pp*-ն՝ միևնույն զանգվածի 8-րդ համարով տարրը, ապա *pp - p = 2*, այսինքն՝ այս դեպքում հանման գործողությունն հանգում է տարրերի համարների հանմանը:

Չանգվածի հետ կապված՝ ցուցչին կարելի է դիմել ինչպես զանգվածի անվանը. օրինակ, եթե ունենք *int a[10]*, **p=a*; ապա *a*[3], *p*[3], **(p+3)* արտահայտություններով դիմվում է զանգվածի միևնույն՝ 3-րդ համարով տարրին:

Միաշափ զանգվածի հետ կապված՝ ցուցչի աշխատանքին ծանոթանալու նպատակով գրենք հետևյալ խնդրի լուծման ծրագիրը. **որոշել 10 տարր պարունակող միաշափ զանգվածի տրված *a* թվին հավասար տարրերի քանակը՝ լուծման գործընթացն իրականացնելով այդ զանգվածի հետ կապված ցուցիչով**:

```

#include <iostream.h>
void main ( )
{ double x[10], *p=0, a;           //1
  int i, l=0;
  cin >> a;
  for (i=0; i<=9; i++) cin >> x[i];
  p=x;                               //2
  for (i=0; i<=9; i++)
  {   if (*p==a) l++;                //3
      p++;                             //4
  } cout << l << endl;
  p -= 10;                             //5
  for (i=0; i<=9; i++)               //6
      cout << p[i] << endl;
}

```

Ծրագրի //1 տողում x զանգվածից բացի հայտարարվել է նաև *double* տիպի p ցուցիչը, որն անմիջապես սկզբնարժեքավորվել է 0 -ով (որպես ազատ ցուցիչ): Այնուհետև զանգվածի տարրերի ներմուծումից հետո //2 տողում p ցուցիչն տրվել է x զանգվածի հասցեն: Այստեղ $p=x$; վերագրումը հնարավոր է, քանի որ զանգվածի x անունը նույնպես *double* տիպի ցուցիչ է (*const* տիպի ցուցիչ՝ զանգվածի 0 համարով առաջին տարրի վրա): //3 տողում ($*p==a$) պայմանով ստուգվում է p -ի ցույց տված տարրի արժեքի հավասարությունը a -ին: Յիկլի կատարման ընթացքում //4 տողում ներառված $p++$; հրամանով p ցուցիչը հերթով ցույց է տալիս զանգվածի տարրերից յուրաքանչյուրի վրա: Յիկլի ավարտին p ցուցիչը «դուրս է գալիս» զանգվածի 10 -րդ տարրին հաջորդող հասցեի վրա: //5-րդ տողում ներառված $p -= 10$; հրամանով ցուցիչը կրկին վերադարձվում է նախկին՝ զանգվածի 0 -ական համարով տարրի վրա, ինչն ունի կհամոզվեք //6-րդ տողում ներառված ցիկլի միջոցով՝ էկրանին արտածելով զանգվածի տարրերը:

Միմվոլային տիպի միաչափ զանգվածն ունի այն առանձնահատկությունը, որ ավարտվում է տողավերջի $'\0'$ պայմանանշանով. ընդ որում՝ եթե անհրաժեշտ է 9 պայմանանշան պարունակող զանգված ունենալ, այն հայտարարում են 10 տարրանոց՝ տեղ պահեստավորելով այդ վերջին ($'\0'$) պայմանանշանի համար ևս: Օրինակ՝ *char s[10]*; հայտարարմամբ տրված տողը կարող է ամենաշատը 9 իմաստալից պայմանանշան պարունակել:

Այս դեպքում ևս զանգվածի անունը հաստատուն ցուցիչ է առաջին՝ 0 -ական համարով տարրի վրա: Օրինակ՝ *char x[4] = {'a', 'b', 'c', '\0'}* զանգվածի x անունը ցուցիչ է $'a'$ պայմանանշանի վրա:

Միմվոլների զանգվածը **տող է**, որն առնում են չակերտների մեջ, օրինակ՝ *“abc”*: Եթե հայտարարենք

```
char *anun[3] = {"Armen", "Karen", "Levon"};
```

սպա *anun[0]*-ն ցուցիչ է, որը ցույց է տալիս *“Armen”*, *anun[1]*-ը՝ *“Karen”*, իսկ *anun[3]*-ը՝ *“Levon”* տողերի վրա:

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ Չնայած ցուցիչների արժեքները թվաբանական արտահայտություններում կարելի է փոփոխել, այդուհանդերձ, զանգվածների անվանումները (որոնք նույնպես ցուցիչներ են) փոփոխման ենթակա չեն:



1. C++-ում զանգվածի անունն ըստ սահմանման ի՞նչ է իրենից ներկայացնում: Կարելի՞ է դրա արժեքը փոփոխել:
2. Քանի՞ հնարավոր տիպի ցուցիչներ գիտեք: Ինչպե՞ս են դրանք հայտարարում:
3. Եթե p -ն փոփոխական ցուցիչ է, որին տրվել է `int a[20]` զանգվածի առաջին տարրի հասցեն, ապա ինչպե՞ս անել, որ այն ցույց փա a զանգվածի՝
 - ա) 3-րդ տարրի վրա, բ) 7-րդ տարրի վրա, գ) 15-րդ տարրի վրա:
4. Ի՞նչ է սիմվոլային զանգվածը. ինչպե՞ս են այն հայտարարում:

§ 2.11 ԳԻՆԱՄԻԿ ՀԻՇՈՂՈՒԹՅՈՒՆ: ՁԱՆԳՎԱԾՆԵՐԻ ՏԵՂԱԿԱՅՈՒՄԸ ԳԻՆԱՄԻԿ ՀԻՇՈՂՈՒԹՅԱՆ ՏԱՐԱԾՔՈՒՄ

Համակարգչում որևէ ծրագիր իրագործելու համար այն տեղակայվում է մեքենայի օպերատիվ հիշողության մեջ և հետո հրաման առ հրաման իրականացվում կենտրոնական պրոցեսորի ղեկավարությամբ: Իրագործվող ծրագրում հայտարարված տվյալները տեղաբաշխվում են օպերատիվ հիշողության, այսպես կոչված, **տվյալների սեզմենտում**, որի ծավալը սահմանափակ է, հիմնականում կազմում է մոտ **64 կբայթ**, որը հաճախ մեծաքանակ տվյալների հետ աշխատող ծրագրերի դեպքում դժվարություններ է հարուցում: Մինչդեռ համակարգչի աշխատանքի համար տրամադրվող հիշողությունը **640 կբայթ** է և կարող է բավարար լինել անգամ մեծաքանակ տվյալների հետ աշխատելու դեպքում: Իրավիճակը փրկում է, այսպես կոչված, **դինամիկ հիշողության** օգտագործումը:

Դինամիկ հիշողությունն օպերատիվ հիշողության 200-ից 300 կբայթ ծավալով այն մասն է, որն ազատ է տվյալների պահպանման համար հատկացված սեզմենտից (64 կբայթ), սրեկային հիշողությունից (16 կբայթ) և ծրագրին հատկացված տիրույթից:

Տվյալների դինամիկ բաշխում ասելով հասկանում են **դինամիկ հիշողության** կիրառումն անմիջապես ծրագրի իրագործման ընթացքում (հիշենք, որ տվյալների սեզմենտում ինֆորմացիան տեղաբաշխվում է ծրագրի թարգմանման փուլում): Տվյալների դինամիկ բաշխման գործընթացի համար հատկանշական է նաև այն, որ մինչև ծրագրի իրագործման պահը հիմնականում հայտնի չեն ոչ դինամիկ հիշողությունում

պահվելիք տվյալների տիպերը, ոչ դրանց ծավալը: Գինամիկ հիշողությունում ստեղծվող տվյալներին չի կարելի անուններով դիմել այնպես, ինչպես թարգմանման փուլում *սրասրիկ կերպով* տեղաբաշխված մեծություններին: Գրանց դիմելու համար կիրառվող հնարավոր միակ միջոցները ցուցիչներն են, որոնք ունակ են հասցե ցույց տալու. չէ՞ որ համակարգչի ողջ հիշողությունը հասցեավորված բջիջներ է ներկայացնում:

Գինամիկ հիշողությունում փոփոխականին տեղ հատկացնելու համար կիրառում են **new** գործողությունը՝ հետևյալ կերպ.

տիպ ցուցիչ = new տիպ;

Օրինակ՝ *int *p = new int;*

Ըստ այս հրամանի՝ համակարգիչը դինամիկ հիշողությունում *int* տիպի մեծություն պահելու համար անհրաժեշտ ծավալով ազատ հիշողություն (*2 բայթ*) է փնտրում և եթե գտնում է, ապա այդ հասցեն տրվում է *p* ցուցիչին, հակառակ դեպքում (եթե դինամիկ հիշողությունում պահանջված քանակությամբ ազատ տարածք չկա)՝ վերադարձվում է *0 (NULL)*:

Գինամիկ հիշողությունում տեղ գտած այն մեծությունները, որոնք, կիրառումից նետո ծրագրի հետագա կատարման ընթացքում այլևս պետք չեն գալու, անհրաժեշտ է «հեռացնել», այլ խոսքով, **դինամիկ հիշողությունն ազատել** ավելորդ մեծություններից: Այդ նպատակով կիրառում են **delete** գործողությունը հետևյալ կերպ՝ *delete p;* :

Գինամիկ հիշողությունում տեղաբաշխվող մեծությունը կարելի է նաև հայտարարման պահին սկզբնարժեքավորել, օրինակ, հետևյալ կերպ՝

*int *p=new int (-6);*

Գինամիկ հիշողությունում կարելի է նաև զանգվածներ պահպանել: Միաշափ զանգվածը դինամիկ հիշողությունում տեղակայելու համար հայտարարում են հետևյալ կերպ՝

տիպ ցուցիչ = new տիպ [զանգվածի տարրերի քանակ];

Օրինակ՝ *double *ps = new double[10];*

Աշխատանքից հետո դինամիկ հիշողությունը զանգվածից ազատելու համար օգտվում են **delete**-ից հետևյալ կերպ՝ *delete [] ps;*

Խ ն դ ի ր . Հաշվել դինամիկ հիշողությունում ստեղծված *n* ամբողջ տիպի տարրեր պարունակող միաջափ զանգվածի դրական տարրերի քանակը:

```
#include <iostream.h>
void main()
{ int i,n,l=0;
  do {cin >> n;} while (n<2 || n>15);
  int *p = new int [n];
  if (p!= NULL) {for (i = 0; i<n; i++) cin >> p [i];
  for (i=0; i<n; i++) if (p[i]>0) l++;
  cout << l << endl;
  delete [ ] p; p = NULL;}
else cout << "դինամիկ հիշողությունում" <<
n << "տարրի համար ազատ տարածք չկա" << endl;
}
```

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ **Գինամիկ հիշողությունում** տեղաբաշխված ավելորդ մեծություններից չազատվելու դեպքում հիշողության կորուստ է առաջանում, որը կարող է ծրագրի կատարման փուլի սխալի հանգեցնել:
- ◆ **Խորհուրդ է** արվում `delete`-ով ազատված ցուցիկ վերագրել `0` կամ `NULL`, քանի որ ազատված և դեռևս կրկին չարժեկրված ցուցի նկատմամբ սխալմամբ կրկին կիրառված `delete`-ը կարող է անկանխատեսելի իրավիճակ ստեղծել:



1. **Ի՞նչ է դինամիկ հիշողությունը, ե՞րբ են այն կիրառում:**
2. **Ի՞նչ են հասկանում տվյալների դինամիկ բաշխում ստելով:**
3. **Որն է մեծության համար դինամիկ հիշողության մեջ ի՞նչ գործողության միջոցով են տեղ հատկացնում:**
4. **Ի՞նչ է վերադարձվում `new` գործողության արդյունքում, եթե դինամիկ հիշողությունում պահանջված ծավալով ազատ տարածք չկա:**
5. **Ինչպե՞ս են դինամիկ հիշողությունն ազատում ավելորդ մեծություններից:**
6. **Հայտարարման փուլում ինչպե՞ս են սկզբնարժեքավորում դինամիկ հիշողությունում տեղակայվող մեծությունը:**
7. **Ինչպե՞ս են դինամիկ հիշողությունում միաչափ զանգված տեղակայում:**
8. **Գինամիկ հիշողությունն ինչպե՞ս են ազատում ավելորդ զանգվածից:**

§ 2.12

ՀԻՇՈՂՈՒԹՅԱՆ ՄԵՋ ՓՈՓՈԽԱԿԱՆՆԵՐԻ ԲԱՇԽՄԱՆ ԱՌԱՆՁՆԱՀԱՏԿՈՒԹՅՈՒՆՆԵՐԸ

Ծրագրում հայտարարվող մեծությունները (փոփոխականները, անվանումով հաստատունները և այլն), բացի տիպից, հատկացված հիշողության ծավալից ու ընդունված արժեքներից, բնութագրվում են նաև

- ա) **հիշողության դասով,**
- բ) **գործողության տիրույթով,**
- գ) **կապակցման ձևով:**

Եթե հետևենք, ապա կարգենք, որ որոշ մեծություններ *կարճատև կյանք ունեն*, որոշներն էլ ծրագրի կատարման ընթացքում մի քանի անգամ ստեղծվում և ոչնչանում են, բայց կան նաև «երկարակյաց» մեծություններ, որոնք ստեղծվելով ծրագրի աշխատանքի սկզբից՝ ուղեկցում են դրան մինչև աշխատանքի ավարտը: Իդենտիֆիկատորի կյանքի տևողությունը բնորոշվում է **հիշողության դասի** միջոցով:

Ծրագրում հայտարարված մեծությունների *հիշողության դասը* բաժանվում է երկու՝ **ավտոմատ** և **ստատիկ** հիշողության դասերի:

Հիշողության **ավտոմատ դասին** են պատկանում այն մեծությունները, որոնք հայտարարվում են բլոկի ներսում. սրանք հայտնի են միայն տվյալ բլոկում և բլոկի սահ-

մաններից դուրս գալուց ոչնչանում են: Նման մեծությունները հայտարարվում են **auto** և **register** առանցքային բառերի միջոցով: Ավտոմատ դասի փոփոխականները կյանքի ժամանակավոր, լոկալ (տեղային) տևողություն ունեն: Օրինակ՝ *auto int k, l*; հայտարարմամբ *k* և *l* ամբողջ տիպի փոփոխականները բացահայտ ձևով որոշվում են որպես ավտոմատ դասի փոփոխականներ:

Քանի որ ցանկացած բլոկում հայտարարված մեծություններն առանց **auto** բնորոշիչի ավտոմատ կերպով դառնում են լոկալ (տեղային), ապա հիմնականում դրանց հայտարարության մեջ **auto** բառը չի նշվում:

register բնորոշիչը կիրառվում է այն դեպքերում, երբ ցանկալի է տվյալ մեծությունը պահպանել համակարգչի արագագործ *ռեգիստրներից* որևէ մեկում: Փոփոխականն իմաստ ունի նման ձևով հայտարարել այն դեպքում, երբ այն կիրառվելու է, օրինակ՝ որպես քանակ կամ գումար հաշվելու միջոց, ցիկլի պարամետր և այլն:

Առանցքային *extern* և *static* բառերով բնորոշում են կյանքի *գլոբալ փոփոխությանը* մեծությունները. նման մեծությունները սկսում են գոյություն ունենալ ծրագրի իրագործման պահից, սակայն սա չի նշանակում, որ դրանք տեսանելի են ամենուրեք: Ծրագրում *գլոբալ օբյեկտը* սահմանվում է միայն մեկ անգամ: Որպեսզի մի քանի ֆայլերից բաղկացած նախագծում բոլոր ֆայլերին տվյալ գլոբալ օբյեկտից օգտվելու հնարավորություն տրվի, այն հայտարարվում է *extern* բառի միջոցով, օրինակ՝ *extern int k*; , որը ցույց է տալիս, որ ֆայլերից որևէ մեկում *int k*; հայտարարություն կա: Իսկ եթե *extern*-ով հայտարարվող գլոբալ օբյեկտը տվյալ պահին սկզբնաբեքավորվում է, օրինակ՝ *extern int k=10*; , ապա *k*-ն համարվում է սահմանված և դրան հենց այդ պահին է հիշողությունում տեղ հատկացվում:

Գոյություն ունեն հիշողության *արարիկ դասին* պատկանող երկու տիպի մեծություններ՝ *արարիկ* և *լոկալ (տեղային)*: Գլոբալ մեծությունների հայտարարությունները տեղակայվում են ցանկացած ֆունկցիայից (նաև *main*-ից) դուրս. դրանք տեսանելի և մատչելի են ծրագրի ցանկացած կետից և այն ֆունկցիաներից, որոնց հայտարարությունները տեղակայված են այդ փոփոխականների հայտարարումից հետո: Գլոբալ փոփոխականները, ինչպես նաև ֆունկցիաների անունները ավտոմատ կերպով ընդունվում են որպես *extern (արարիկ)*: Գլոբալ մեծություններն իրենց արժեքները պահպանում են ծրագրի կատարման ողջ ընթացքում:

Ստատիկ (*static*) հայտարարվում են *լոկալ (տեղային) նշանակության* մեծությունները, որոնք հայտնի են միայն այն բլոկում, ուր հայտարարված են: Չնայած այս փոփոխականները «դրսից» տեսանելի չեն, սակայն ամեն անգամ, երբ ղեկավարումը տրվում է այն բլոկին, ուր դրանք հայտարարված են, *static* փոփոխականները վերականգնում են իրենց վերջին անգամ ստացած արժեքները: Եթե ստատիկ փոփոխականը հայտարարվելիս չի սկզբնաբեքավորվում, ապա մեքենան դրան ավտոմատ կերպով նախնական *0* արժեք է տալիս:

Հայտարարված մեծության *գործողության փիրույթը* կամ, ինչպես հաճախ են ասում, *փեսանելիության փիրույթը* ծրագրի այն հատվածն է, որտեղ այն հասանելի է, այլ խոսքով՝ որտեղ կարելի է օգտագործել:

C++-ում հայտարարված մեծությունների գործողության տիրույթի 4 տիպեր կան՝ **ֆունկցիա**, **ֆայլ**, **բլոկ** և **ֆունկցիայի նախարիպ**:

Այն մեծությունները, որոնց հայտարարությունները կատարվել են ծրագրում առ-

կա ֆունկցիաներից դուրս, ունեն **ֆայլ գործողության տիրույթ**: Դրանց կարելի է դիմել իրենց հայտարարման մասից սկսած՝ ֆայլում առկա բոլոր ֆունկցիաներից:

Նշիչները այն միակ իդենտիֆիկատորներն են, որոնց **գործողության տիրույթը ֆունկցիան** է. նշիչները հայտնի են միայն այն մարմնում, որտեղ նշվում են և դրանցից դուրս դառնում են «անտեսանելի»: Նշիչներ օգտագործվում են ինչպես *switch* օպերատորում՝ որպես *case*-ի արժեքներ, այնպես էլ անցման (*goto*) օպերատորում:

Բլոկի ներսում (ձևավոր { } փակագծերի մեջ) հայտարարված մեծությունների համար **գործողության տիրույթը** բլոկն է, որից դուրս դրանց հնարավոր չէ դիմել:

Գործողության **ֆունկցիայի նախապիս** տիրույթին կծանոթանանք ֆունկցիաներն ուսումնասիրելիս (§ 2.13):

Կապակցման ձևը որոշում է, թե տվյալ մեծությունը հայտնի է միայն ընթացիկ ծրագրում, թե՞ նախագծի բաղկացուցիչ ֆայլերում ևս:

Հետևենք հետևյալ ծրագրի աշխատանքին.

```
# include <iostream.h>
int k=5;
void main ()
{ int k=3;
cout << "Sa main-i local k-n e" << k << endl;
cout << "Sa global k-n e" << :: k << endl;
    { int k=7;
      cout << "Sa bloki local k-n e" << k << endl;
    }
    cout << "Sa main-i local k-n e" << k << endl;
}
```

Արդյունքում էկրանին կարտածվի հետևյալ ինֆորմացիան.

Sa main-i local k-n e 3

Sa global k-n e 5

Sa bloki local k-n e 7

Sa main-i local k-n e 3

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ Չի կարելի միևնույն իդենտիֆիկատորը մեկից ավելի հիշողության դասի տիրույթում բնորոշել, օրինակ, *register*-ից բացի տալ նաև *auto*:
- ◆ *register* առանցքային բառը կարող է կիրառվել միայն լոկալ փոփոխականների նկատմամբ:
- ◆ Ավտոմատ հիշողության դասի միջոցով հիշողություն է փոփոխվում, քանի որ բլոկից դուրս գալուց նման հիշողությունն ազատվում է:



1. Իդենտիֆիկատորի տիպից և հատկացվող հիշողության ծավալից բացի՝ այն բնութագրող ուրիշ ի՞նչ հատկանիշներ գիտեք:
2. Որո՞նք են իդենտիֆիկատորին հատկացվող հիշողության դասի չեզ հայտնի բնութագրիչները:
3. Ի՞նչ է նշանակում ծրագրում հայտարարված մեծությունների գործողության կամ տեսանելիության տիրույթ հասկացությունը:
4. Ի՞նչ է որոշում հայտարարված մեծությունների կապակցման չեք:
5. Ո՞ր մեծություններն են C++-ը առանց մեր միջամտության բնորոշում ավտոմատ հիշողության դասի տիպով:
6. Ո՞ր մեծություններն են կոչվում գլոբալ:
7. Ո՞ր մեծություններն են կոչվում լոկալ:
8. Ո՞ր մեծություններն են հայտարարվում որպես static:

§ 2.13 ՖՈՒՆԿՑԻԱՆԵՐ: ՖՈՒՆԿՑԻԱՆԵՐԻ ՑՈՒՑԵՉՆԵՐ

Ֆունկցիայի հասկացությունը C++-ի հիմնային հասկացություններից է. նույնիսկ C++ ծրագրի առաջնային, գլխավոր միավոր հանդիսացող *main ()* կառույցն է ձևակերպվում որպես ֆունկցիա: Կարելի է ասել, որ

Ֆունկցիան որոշակի իմաստով ինքնուրույն ծրագրային միավոր է, որը գրվելով մեկ անգամ՝ հնարավոր է կիրառել բազմիցս:

Ֆունկցիան պետք է **սահմանված** կամ **նկարագրված** լինի ավելի վաղ, քան դրա կիրառումը (կանչը): Ֆունկցիայի սահմանումն ունի հետևյալ ընդհանուր տեսքը.

*Ֆունկցիայի տիպ ֆունկցիայի անուն (ֆունկցիայի պարամետրեր)
ֆունկցիայի մարմին*

Ֆունկցիայի տիպը ֆունկցիայից վերադարձվող արժեքի տիպն է: Եթե ֆունկցիան արժեք չի վերադարձնում, ապա դրա տիպը նկարագրվում է **void** առանցքային բառով: Օրինակ՝ `void min(int k)`, կամ `double kk(int c)` հայտարարությունները ֆունկցիայի ճիշտ վերնագրեր են:

Ֆունկցիայի անունը ցանկացած իդենտիֆիկատոր է: C++-ում ֆունկցիայի անվանը հիշողության *extern* կարգ է վերագրվում, և լինելով *գլոբալ*՝ որոշակի պայմանների դեպքում ֆունկցիան ամենուրեք մատչելի է այն ծրագրային մոդուլի շրջանակներում, որտեղ սահմանված է:

Ֆունկցիայի պարամետրերն իրարից ստորակետերով փոխանցատված այն մե-

ծություններն են (**Ֆորմալ պարամետրերը**), որոնց արժեքները ֆունկցիան ստանում է կանչի պահին. սրանք կոչվում են նաև **մուտքային պարամետրեր**:

Եթե ֆունկցիային պարամետրեր չեն փոխանցվում (պարամետրերի ցանկը դատարկ է), ապա հնարավոր պարամետրերի փոխարեն գրվում է **void** առանցքային բառը կամ այդ մասը թողնվում է դատարկ՝ (): Օրինակ՝ `int max()` և `int max(void)` վերնագրերն իրար համարժեք են:

Եթե ֆունկցիայի վերնագրում պարամետրեր կան, ապա դրանք կարող են հայտարարվել հետևյալ երկու եղանակներով՝

ա) *տիպ պարամետրի անուն*,

բ) *տիպ պարամետրի անուն = լռությամբ փրվող արժեք*:

Եթե ֆունկցիայի վերնագրում պարամետրերը թվարկվում են ա) տարբերակով, ապա տվյալ ֆունկցիայի կանչը պետք է պարունակի այնքան պարամետր (**փաստացի պարամետր**), որքան պարամետր կա նկարագրված. բ) տարբերակի դեպքում հնարավոր է թե՛ առկա պարամետրերի քանակին համապատասխան փաստացի պարամետրերով կանչ ունենալ և թե՛ միայն այնքան փաստացի պարամետր տալ, որքան որ լռությամբ չարժևորված ֆորմալ պարամետրեր կան ֆունկցիայի վերնագրում:

Ֆունկցիայի մարմինը բլոկ է, այսինքն՝ { } փակագծերում ներառված նկարագրությունների, հայտարարությունների և օպերատորների ցանկացած հաջորդականություն:

Ֆունկցիայի նկարագրումը կատարվում է **ֆունկցիայի նախադիպի** տրմամբ:

Ֆունկցիայի նախադիպը ֆունկցիայի վերնագրի նմանակն է, որտեղ կարող են բացակայել ֆորմալ պարամետրերի անունները, բայց պարպադիր կերպով նշվում են դրանց տիպերը:

Եթե նախատիպում ֆունկցիայի պարամետրերը նկարագրվում են տիպերով և համապատասխան իդենտիֆիկատորներով, ապա դրանք այն միակ իդենտիֆիկատորներն են, որոնց **գործողության փիրույթը ֆունկցիայի նախադիպն է**. Քարգմանիչը ֆունկցիայի նախատիպում առկա պարամետրերի անվանումներն անտեսում է: Նախատիպը, ի տարբերություն ֆունկցիայի վերնագրի, ավարտվում է ;-ով: Եթե ֆունկցիան նկարագրվում է նախատիպի միջոցով, ապա ամբողջ ֆունկցիան կարելի է սահմանել արդեն այն կիրառող ծրագրային մոդուլի մարմնի վերջում: Ֆունկցիան նախատիպերով տալը մեծացնում է ծրագրի հուսալիությունը, քանի որ թարգմանիչ ծրագիրը խստորեն «պահանջում է», որ ֆունկցիայի կանչի պահին փոխանցվող արգումենտները (**փաստացի պարամետրերը**) տիպերով ու քանակությամբ համընկնեն ոչ միայն ֆունկցիայի նախատիպի, այլև ֆունկցիայի վերնագրում առկա ֆորմալ պարամետրերի տիպերին ու դրանց քանակին: Այս համապատասխանությունը ստուգվում է թարգմանման փուլում, և եթե նախատիպի հայտարարություն չլինի և ֆունկցիայի կանչն էլ նախորդի ֆունկցիայի սահմանմանը, ապա կհայտարարվի սխալի՝ անհայտ մեծության կանչի մասին: Դա է պատճառը, որ կիրառվող ֆունկցիաների նախատիպերը տրվում են ծրագրի սկզբում: Կարելի է նաև նախապես

ֆունկցիաների նախատիպերից բաղկացած վերնագրային ֆայլ ձևավորել և այն `#include`-ի միջոցով կցել ծրագրին:

Ֆունկցիայի ճիշտ նախատիպ է, օրինակ՝ հետևյալը.

```
int min(double, int);
```

Ֆունկցիայի մարմնում հայտարարված մեծությունները և վերնագրում հայտարարված պարամետրերը լոկալ են (պեդային) և ֆունկցիայից դուրս հայտնի չեն:

Եթե ֆունկցիայի տիպը `void` չէ, ապա ֆունկցիայի մարմնում գոնե մեկ անգամ պետք է հանդիպի `return` ծառայողական բառը, որին հաջորդող արտահայտության արժեքն էլ հանդիսանում է ֆունկցիայից վերադարձվող արժեքը, որի տիպը պետք է համընկնի ֆունկցիայի տիպի հետ: Ընդ որում՝ եթե անգամ ֆունկցիայի մարմնում `return`-ը բազմաթիվ անգամ է հանդիպում, ֆունկցիայի աշխատանքն ավարտվում է իրագործվող առաջին `return`-ով:

Ֆունկցիայի կանչը մի արտահայտություն է, ուր ֆունկցիայի անվան հետ միասին փակագծերում () թվարկվում են ֆունկցիային փոխանցվող մեծությունները, որոնք կոչվում են **փաստացի պարամետրեր**: Եթե ֆունկցիայի ֆորմալ պարամետրերի ցուցակը դատարկ է, ապա դրա կանչի պահին () փակագծերում ոչինչ չի գրվում, սակայն փակագծերը պարտադիր են: Չպետք է մոռանալ, որ ֆունկցիայի կանչում ներառված փաստացի պարամետրերի տիպերը պետք է համընկնեն ֆունկցիայի վերնագրում համապատասխան դիրքերում եղած ֆորմալ պարամետրերի տիպերի հետ:

Օրինակ՝ որոշենք տրված `a`, `b`, `c` իրարից տարբեր իրական մեծություններից մեծագույնի արժեքը՝ 3 պարամետրերից մեծագույնը որոշող ֆունկցիայի կիրառմամբ:

```
#include <iostream.h>
double max(double, double, double) ; //ֆունկցիայի նախատիպ
void main()
{
    double a,b,c ;
    cout << "a=" ; cin >> a ;
    cout << "b=" ; cin >> b ;
    cout << "c=" ; cin >> c ;
    cout << max(a,b,c) << endl ; // առաջին կանչ
    cout << max(b,a,10.7) << endl ; // երկրորդ կանչ
    cout << max(b,15.8,a) << endl ; // երրորդ կանչ
}
double max(double x, double y, double z) // ֆունկցիայի վերնագիր
{
    double m ;
    if (x>y) m=x ; else m=y ;
    if (z>m) return z; //1
    else return m; //2
}
```

Բերված ծրագրում `#include`-ին հաջորդող տողում նախ բերվել է ֆունկցիայի նախատիպը, ըստ որի ֆունկցիան դրսից պետք է 3 իրական տիպի պարամետրեր ընդունի և մեկ իրական արժեք վերադարձնի: `main()`-ում `a`, `b` և `c` մեծությունների ներմու-

ծումից հետո //առաջին կանչ մակագրությամբ տողում կանչ է կատարվել ֆունկցիային, որին փոխանցվել է *a*-ն, որը ֆունկցիան ընդունել է *x* ֆորմալ պարամետրի մեջ, *b*-ն, որն ընդունել է *y*-ի մեջ, և *c*-ն, որն ընդունել է *z*-ում (նայել // *ֆունկցիայի վերնագիր* տողը): Ֆունկցիային երկրորդ անգամ կանչելիս, որպես *x*, *y* և *z* ֆորմալ պարամետրերի արժեքներ, համապատասխանաբար ուղարկվել են *b*, *a* և *10.7*, իսկ երրորդ կանչի դեպքում՝ *b*, *15.8* և *a* մեծությունները: Ֆունկցիայի սահմանումը կատարվել է *main()*-ի մարմնից հետո՝ սկսած //ֆունկցիայի վերնագիր մակագրությամբ տողից. այստեղ //1 տողում գրված *return*-ը կավարտի ֆունկցիայի աշխատանքը, եթե պարզվի, որ *z > m* պայմանն ունի *true* արժեք, հակառակ դեպքում ֆունկցիան աշխատանքը կավարտի //2 տողում եղած *return*-ով:

Ընդհանրապես *ֆունկցիայի կանչը ֆունկցիայից վերադարձվող արժեքի տիպն ունեցող արտահայտություն է*: Եթե ֆունկցիայի տիպը *void* չէ, ապա այդ ֆունկցիայի կանչը կարելի է կիրառել տվյալ արժեքի տիպի համար թույլատրելի ցանկացած արտահայտության մեջ:

C++-ում *ֆունկցիայի անունը ցուցիչ է հիշողության այն հասցեի վրա, որտեղից սկսած տեղավորված է ֆունկցիայի մարմինը*. այս առումով ֆունկցիայի և զանգվածի անուններն իրար նման են:

Այսպիսով, ֆունկցիայի անվան պարունակած հասցեն կարելի է վերագրել մեկ այլ՝ ֆունկցիայի տիպն ունեցող ցուցիչի և կիրառել տվյալ ֆունկցիան հետագայում կանչելու համար:

Ֆունկցիայի վրա ցույց տալու նպատակով ստեղծվող ցուցիչը հայտարարում են հետևյալ կերպ.

*տիպ (*ցուցիչ անուն)(ֆունկցիայի պարամետրերի տիպերը);*

Օրինակ՝ *int (*p)(double)*; արտահայտությամբ հայտարարված է *fp* ցուցիչ, որը «կկարողանա» ցույց տալ *double* տիպի ֆորմալ պարամետրով և *int* տիպի արժեք վերադարձնող ցանկացած ֆունկցիայի վրա. ցուցիչն ընդգրկող () փակագծերը պարտադիր են՝ այլապես հայտարարվածը *int* տիպի ցուցիչ վերադարձնող *fp* անունով ֆունկցիայի նախատիպի սահմանում կլիներ:

Օրինակ՝

```
#include <iostream.h>
int f1(int a, int b) {return a+b;}
int f2(int x, int y) {return x-y;}
int f3(int c, int d) {return c*d;}
void main ()
{ int (*p)(int, int);           //1
  p=f1;                         //2
  cout << (*p)(4,5) << endl;    //3
  p=f2;                         //4
  cout << (*p)(4,5) << endl;    //5
  p=f3;                         //6
  cout << (*p)(4,5) << endl;    //7
}
```

Այս ծրագրի աշխատանքի ընթացքում նախ հայտարարվում է p ցուցիչ ($//1$), որն ըստ հայտարարման ձևի կարող է ցույց տալ երկու *int* տիպի ֆորմալ պարամետրերով և *int* տիպի արժեք վերադարձնող ֆունկցիայի վրա: Այնուհետև $p=f1$; ($//2$) վերագրմամբ p -ին տրվում է $f1$ ֆունկցիայի հասցեն: $//3$ -րդ տողում կանչվում է $f1$ ֆունկցիան, որի վրա տվյալ պահին ցույց է տալիս p ցուցիչը (նկատենք, որ եթե կանչը ձևակերպվի $*p(4,5)$ տեսքով՝ քերականական սխալ կլինի), այսպիսով, $//3$ տողում առկա կանչի արդյունքում էկրանին կարտածվի $4+5$ -ի արժեքը: Այնուհետև $//4$ տողում p ցուցիչը $p=f2$; վերագրմամբ ստանում է $f2$ ֆունկցիայի հասցեն և $//5$ -րդ տողում կայացած ֆունկցիայի կանչի շնորհիվ արտածվում է $4-5 = -1$ թիվը: Վերջում $//6$ տողով p -ն ստանում է $f3$ ֆունկցիայի հասցեն և $//7$ տողում կայացած ֆունկցիայի կանչի արդյունքում էկրանին արտածվում է $4*5 = 20$ թիվը:

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ **Ֆունկցիայի վերնագրում միևնույն փիպն ունեցող պարամետրերը չի կարելի համախմբել այդ փիպն արտահայտող բառի փակ անկախ ամեն ինչից, յուրաքանչյուր պարամետր պետք է բնութագրվի իր փիպով:**
- ◆ **Ֆունկցիան սահմանելիս վերնագրից հետո չի կարելի կեդր-ստորակետ (:) դնել. դա քերականական սխալ է:**
- ◆ **Թյուրիմացությունից խուսափելու նպատակով խորհուրդ է փոխվում իրար համապատասխանող ֆորմալ և փաստացի պարամետրերը նույն անուններով չկոչել՝ թեպետ դա սխալ չէ:**
- ◆ **Գաղարկ պարամետրերով ֆունկցիային կանչելիս փակագծեր () չնշելը չի դիպվում որպես քերականական սխալ, սակայն այդ դեպքում ֆունկցիայի կանչ չի իրականացվում:**
- ◆ **Ֆունկցիան պետք է երկար գրված մարմին չունենա (ամենաշարք՝ էկրանի կամ դրա կեսի չափով), չնայած լեզուն սահմանափակում չի դնում:**
- ◆ **Եթե ֆունկցիան void փիպի չէ, այսինքն՝ փիպ ունի, բայց return չի պարունակում մարմնի մեջ, ապա քերականական սխալ է:**
- ◆ **Ֆունկցիայի անունը պետք է լինի այնպիսին, որ «հուշի» ֆունկցիայի իմաստը:**



1. **Ֆունկցիա հայտարարելու քանի՞ եղանակ գիտեք:**
2. **Կյանքի ի՞նչ փնտրություն ունեն ֆունկցիայում հայտարարված մեծությունները. կարելի՞ է ֆունկցիայից դուրս դրանք կիրառել:**
3. **Հիշողության ի՞նչ կարգ է վերագրվում ֆունկցիայի անվանը:**
4. **Ինչպե՞ս են անվանում ֆունկցիայի վերնագրում և ֆունկցիայի կանչում եղած մեծությունները:**
5. **Ինչպե՞ս է հայտարարվում այն ֆունկցիան, որն արժեք չի վերադարձնում:**
6. **Ինչի՞ համար է return-ը կիրառվում. քանի՞ return կարող է պարունակել ֆունկցիան:**
7. **Ի՞նչ է ֆունկցիայի նախափիպը, ո՞րն է դրա իմաստը:**

8. Ի՞նչ տարբերություններ կան *ֆունկցիայի վերնագրի ու դրա նախադիպի միջև*:
9. *Կազմել և օգտագործել մի ֆունկցիա, որը կվերադարձնի a, b, c, d մեծություններից փոքրագույնի արժեքը*:
10. *Ֆունկցիայի անունն ըստ սահմանման ի՞նչ է*:
11. *Ինչպե՞ս են հայտարարում ֆունկցիայի վրա ցույց տալու համար նախադասված ցուցիչը*:
12. *Ֆունկցիայի վրա ցույց տվող ցուցչի միջոցով կանչել f1 ֆունկցիային, որը double տիպի պարամետրի քառակուսի է վերադարձնում և f2 ֆունկցիային, որը double տիպի պարամետրի խորանարդ ստորինասի արժեք է վերադարձնում*:

ՆԵՐԿԱՌՈՒՅՎՈՂ (inline) ՖՈՒՆԿՑԻԱՆԵՐ:

§ 2.14

ՖՈՒՆԿՑԻԱՆԵՐԻՑ ԱՐԺԵՔՆԵՐ ՎԵՐԱԳԱՐՉՆԵԼՈՒ ԱՅԼ ԵՂԱՆԱԿՆԵՐ

Համակարգիչը ֆունկցիայի կանչն իրագործելիս ծրագրի վիճակի վերաբերյալ այդ պահին եղած ինֆորմացիան (ֆունկցիայի կանչին հաջորդող հրամանի հասցեն, ռեգիստրների պարունակությունները և այլն) պահպանում է հիշողության մեկ այլ տարածքում՝ *սպեկում*, իսկ օպերատիվ հիշողությունը տրամադրում է կանչվող ֆունկցիային: Աշխատանքի ավարտին ֆունկցիան ազատում է օպերատիվ հիշողությունը, իսկ ծրագրի վերաբերյալ ստեղծված պահպանված ինֆորմացիան նորից բեռնավորվում է օպերատիվ հիշողություն: Բնականաբար, ֆունկցիայի կիրառումը ծրագրի աշխատանքը դանդաղեցնում է: Երբ ժամանակի գործոնն էական է, իսկ կիրառված ֆունկցիան՝ փոքրածավալ, օգտվում են, այսպես կոչված, *ներկառուցված ֆունկցիայից*: Այս դեպքում ֆունկցիայից վերադարձվող արժեքը բնութագրող տիպից առաջ գրվում է *inline* առանցքային բառը: Ծրագրի թարգմանման ու կապակցման փուլում նման ֆունկցիային առնչվող բոլոր կանչերը փոխարինվում են տվյալ ֆունկցիայի մարմնի պատճենով՝ նախօրոք ֆորմալ պարամետրերը փոխարինելով տվյալ կանչին համապատասխանող փաստացի պարամետրերի արժեքներով: Որպեսզի ֆունկցիայի ներկառուցումից հետո ծրագրի իրագործման ենթակա կողը չափազանց ծավալուն չստացվի՝ խորհուրդ է տրվում *inline* հայտարարել միայն շատ փոքրածավալ կամ քիչ կանչեր ունեցող ֆունկցիաները: *inline* ֆունկցիան սահմանվում է առանց նախատիպի տրման՝ ամբողջությամբ:

Օրինակ. *ներկառուցվող ֆունկցիայով որոշել իրարից տարբեր a և b պարամետրերից փոքրագույնի արժեքը*:

```
#include <iostream.h>
inline double min(double a, double b)           // 0
{ return a<b ? a : b;}
void main()
```

```

{ double x,y ;
  cout << "x=" ; cin >> x ;
  cout << "y=" ; cin >> y ;
  cout << min(x,y) << endl ; // առաջին կանչ
  cout << min(x,9.8) << endl ; // երկրորդ կանչ
}

```

Այստեղ //0 մեկնաբանությամբ տողում *min* ֆունկցիան հայտարարվել է որպես ներկառուցվող: Այդ պատճառով //առաջին կանչ տողում կատարված կանչը ծրագրի թարգմանման փուլում փոխարինվում է $x < y ? x : y$ արտահայտությամբ (a -ն և b -ն փոխարինվել են փաստացի x և y պարամետրերով), իսկ //երկրորդ կանչ տողում՝ $x < 9.8 ? x : 9.8$ գրառմամբ:

Ծանոթանանք ֆունկցիայից արժեք վերադարձնելու այլ եղանակների հետ:

Ընդհանրապես ծրագրավորման լեզուներում տարբերում են *ֆունկցիային դիմելու* երկու եղանակ՝ *արժեքով կանչ* և *հղումով կանչ*: Երբ ֆունկցիան կանչվում է ըստ *արժեքի*, այդ դեպքում ֆունկցիային փոխանցվում են *փաստացի պարամետրերի պատճենները*: Վերը քննարկած բոլոր դեպքերում ֆունկցիաները կանչվել են ըստ արժեքի: Բնական է, որ ֆունկցիայում *պարամետրերի պատճենների հետ կատարված փոփոխությունները չեն ազդում կանչող ծրագրում դրանց համապարաստակա՞նող փաստացի պարամետրերի արժեքների վրա*: Ֆունկցիային դիմելու արժեքով կանչի եղանակի վատ կողմն այն է, որ մեքենան լրացուցիչ ժամանակ է ծախսում պատճեններ ստեղծելիս, առավել ևս՝ եթե փոխանցվող պարամետրերը մեծաքանակ են:

Ֆունկցիայի կանչի մյուս՝ *հղումով* եղանակի դեպքում փաստացի պարամետրերի *հղումներն* են փոխանցվում ֆունկցիային: Հիշենք, որ հղումները, լինելով միևնույն փոփոխականի երկրորդ անունը, «տեղյակ» են հիշողությունում դրանց գտնվելու հասցեից: Այսպիսով, *հղումով փոխանցված փոփոխականի հետ ֆունկցիայում կատարված փոփոխություններն իրականացվում են հենց փաստացի պարամետրերի հետ*. ստացվում է, որ ֆունկցիան առանց լրացուցիչ միջոց կիրառելու, հղման միջոցով նկարագրված փոփոխականով արժեք «վերադարձրեց»:

Դիտարկենք հետևյալ խնդիրը. *ֆունկցիայի միջոցով հաշվել a և b փոփոխականների գումարը՝ արժեքը վերադարձնելով հղմամբ*:

```

#include <iostream.h>
void gumar(double, double,double&);           // 0
void main()
{ double a,b,gum ;
  cout << "a=" ; cin >> a;
  cout << "b=" ; cin >> b;
  gumar(a,b,gum); //1
  cout << gum << endl;
  gum= 100;                                       //11
  gumar(7.25,6.2,gum);                           //2
  cout << gum << endl;
}

```

```
void gumar(double x, double y, double &s)    // 3
{
    s=x+y;
}
```

Ինչպես երևում է //0 տողում եղած ֆունկցիայի նախատիպից, առաջին երկու պարամետրերը ֆունկցիան կվերցնի որպես արժեք պարամետրեր (կատեղծվեն փոխանցվող փաստացի պարամետրերի պատճենները), իսկ երրորդ պարամետրը կփոխանցվի հղմամբ (կատեղծվի համապատասխան պարամետրի երկրորդ անունը՝ *s*): Ծրագրի //1 տողում կատարված ֆունկցիայի կանչից հետո *gum*-ը կպարունակի $a + b$ -ի արժեքը: Գրան հաջորդող //11 տողում այդ արժեքը փոխվում է 100-ով, սակայն //2 կանչի արդյունքում կհամոզվենք, որ այն դարձել է հավասար 13.45-ի, այսինքն՝ ֆունկցիան փոխել է *gum*-ի արժեքը:

Եթե ցանկալի է, որ պարամետրը փոխանցվի հղմամբ, բայց ֆունկցիան «չկարողանա» փոխել փաստացի պարամետրի արժեքը, ապա անհրաժեշտ է այն *const* տիպի ֆորմալ պարամետր հայտարարել: Բերված օրինակում ֆունկցիայի վերնագիրը այս դեպքի համար կունենար հետևյալ տեսքը՝

```
void gumar (double, double, const double&);
```

Քանի որ բերված խնդրում *s*-ի արժեքը փոխվում է, ապա այն չէր կարող *const*՝ հաստատուն լինել: Ֆունկցիայից արժեքներ վերադարձնելու երրորդ եղանակը ***ցուցիչներ*** կիրառումն է, որը նույնպես հղումային եղանակ է:

Գիտարկենք հետևյալ խնդիրը. ***ֆունկցիայի միջոցով որոշել x իրական պարամետրի խորանարդը՝ արժեքը վերադարձնելով ցուցիչով:***

```
#include <iostream.h>
void cube(double, double *);           // 1
void main()
{
    double a,aa;
    cin >> a ;
    cube(a,&aa) ;                       //2
    cout << aa <<endl;
    cube(3,&aa) ;                       //3
    cout << aa <<endl;
}
void cube(double x, double *p)
{
    *p = x * x * x;
}
```

Ինչպես երևում է //1 տողում ֆունկցիայի նախատիպից՝ ֆունկցիան կվերցնի փոխանցված առաջին պարամետրի պատճենը և երկրորդ պարամետրի հասցեն:

//2 տողում իրականացվել է ֆունկցիայի առաջին կանչը, որի դեպքում դրան են փոխանցվել *a*-ի արժեքն ու *aa*-ի հասցեն: //3 տողով ֆունկցիան երկրորդ անգամ է կանչվել՝ այս անգամ 3-ի խորանարդը հաշվելու համար, որը նորից ստացվել է *aa*-ի մեջ:

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ *Inline* ֆունկցիայի կիրառումը կարող է կրճատել ծրագրի իրագործման ժամանակահատվածը, բայց ավելացնել ծրագրի ծավալը:
- ◆ Կոմպիլյատորը հաճախ անհրաժեշտության դեպքում որոշ ֆունկցիաներ ավտոմատ համարում է *inline*, իսկ որոշ դեպքերում հաշվի չի առնում ծրագրավորողի կողմից արված *inline* հրահանգը:
- ◆ Եթե ֆունկցիան հղումով է կանչվում, սպա փոխանցվող մեծաքանակ արվյալները պարճենելու վրա ավելորդ ժամանակ ծախսելուց չեղբազարվում ենք:
- ◆ Հղումով կանչը բուլացնում է արվյալների պաշտպանությունը և այն անհարկի պեղք չէ օգտագործել:
- ◆ Հղումով փոխանցված պարամետրը ֆունկցիայի մարմնում կիրառվում է միայն անունով՝ առանց ամպերսանդի (&). պեղք է զգույշ լինել այն չշփոթելու համար որպես արժեք փոխանցված պարամետրի հեղք:
- ◆ Խորհուրդ է արվում ֆունկցիայից արժեք վերադարձնող պարամետրերը հայտարարել որպես ցուցիչ, չփոփոխվող պարամետրերը՝ որպես արժեք պարամետր, իսկ այն ծավալուն պարամետրերը, որոնք չպեղք է փոփոխվեն՝ որպես հաստատուն հղումներ:



1. Ո՞ր ֆունկցիաներն է խորհուրդ արվում հայտարարել *inline*:
2. Ի՞նչ է կատարվում ներկառուցված ֆունկցիայի հեղք ծրագիրը քարզմաննելիս:
3. Ֆունկցիայից արժեքներ վերադարձնելու քանի՞ եղանակ գիղք:
4. Ֆունկցիային փոխանցված *a* և *b* պարամետրերի գումարը վերադարձնել հղման, իսկ արդարոյալը՝ ցուցչի միջոցով:

§ 2.15 ԶԱՆԳՎԱԾՆԵՐԻ ՓՈԽԱՆՑՈՒՄԸ ՖՈՒՆԿՑԻԱՆԵՐԻՆ: ՖՈՒՆԿՑԻԱՆԵՐԻ ՎԵՐԱԲԵՌՆԱՎՈՐՈՒՄԸ

Ֆունկցիային զանգված փոխանցելու նպատակով անհրաժեշտ է ֆունկցիայի կանչի մեջ նշել դրա անունն ու տարրերի քանակը: Օրինակ՝ *ff* ֆունկցիային *double x[20]* զանգվածը փոխանցելու համար բավարար է գրել հետևյալը՝ *ff(x,20)*: Ֆունկցիայի վերնագիրն էլ, իր հերթին, զանգված ընդունելու համար պետք է համապատասխան ֆորմալ պարամետրեր պարունակի: Օրինակ՝ վերը բերված *ff* ֆունկցիայի համար ճիշտ վերնագիր կարող էր լինել հետևյալ գրառումը՝ *void ff(double y[],int k)*, որտեղ *y*-ը զանգվածն «ընդունելու» համար նախատեսված ֆորմալ պարամետրն է, իսկ *k*-ն՝ զանգվածի տարրերի քանակը:

**C++ լեզուն զանգվածի փոխանցումն ավտոմատ
իրականացնում է հղմամբ:**

Ստանալով հիշողությունում զանգվածի տեղաբաշխման առաջին հասցեն՝ ֆունկցիան հնարավորություն է ստանում այն փոփոխելու: Ի տարբերություն ամբողջ զանգվածի, ցանկացած առանձին տարր կարելի է փոխանցել ըստ արժեքի, եթե ֆունկցիայի կանչի մեջ որպես փաստացի պարամետր տրվող զանգվածի անվանը կից քառակուսի փակագծերում նշվի տվյալ տարրի համարը: Օրինակ՝ *ff(x[10])* կանչի արդյունքում *ff* ֆունկցիային որպես փաստացի պարամետր կփոխանցվի *x* զանգվածի 11-րդ տարրը:

Քննարկենք հետևյալ ծրագիրը.

```
#include <iostream.h>
void zang_fofoxum (int[ ],int);           //1
void main()
{
    const int n=10;                       //տարրերի քանակը
    int x[n]={9,8,7,6,5,4,3,2,1,0};
    zang_fofoxum(x,n);                   //2
    for(int i=0;i<n,i++)                 //3
        cout <<x[i] <<endl;
}
void zang_fofoxum(int y[ ],int m)       //4
{
    int i,k;
    for (i=0 ;i<m/2;i++)
        { k=y[i];
          y[i] = y[m - i - 1];
          y[m - i - 1] = k;
        }
}
```

Ծրագրի //1 մակագրությամբ տողում հայտարարվել է `zang_fofoxum` ֆունկցիայի նախատիպը, ըստ որի ֆունկցիային պետք է միաչափ զանգված փոխանցվի: Ծրագրում սահմանված 10 տարրեր պարունակող զանգվածի տարրերը սկզբնարժեքավորվել են 9, 8, ..., 1, 0 ամբողջ թվերով: Այնուհետև (//2) կանչ է կատարվել ֆունկցիային, որը սահմանվել է //4 տողից սկսած. այստեղ ավտոմատ կերպով հղմամբ փոխանցվող զանգվածի համար ստեղծվել է երկրորդ y անունը: Ըստ ֆունկցիայի ալգորիթմի՝ զանգվածը շրջվել է և ստացվել նոր զանգված $y[0]=0$; $y[1]=1$; ..., $y[9]=9$ տարրերով:

Ֆունկցիայի աշխատանքի ավարտից հետո ծրագրի //3 տողում առկա ցիկլի միջոցով արտածվել է x զանգվածը, որի տարրերը, կպարզվի, որ նույնպես շրջվել են: Այսպիսով, համոզվեցինք, որ զանգվածը, առանց մեր միջամտության, ֆունկցիային փոխանցվել է ոչ թե որպես արժեք պարամետր, այլ՝ հղմամբ:

Երբեմն նպատակահարմար է լինում մի քանի ֆունկցիաներ անվանել միևնույն անունով: `C++` լեզուն նման հնարավորություն ընձեռում է և այդ գործընթացը կոչվում է **ֆունկցիաների վերաբեռնավորում**: Վերաբեռնավորվող ֆունկցիաները միևնույն անուն են կրում, սակայն պետք է ունենան պարամետրերի տարբեր հավաքածուներ, այսինքն՝ պետք է *տարբեր քանակությամբ, տարբեր տիպերի պարամետրեր ունենան*: Որպեսզի համակարգիչը կարողանա հստակորեն կողմնորոշվել, թե նույն անունը կրող ֆունկցիաներից ո՞ր մեկն է տվյալ պահին կանչվում՝ ստուգում է պարամետրերի քանակը, դրանց տիպերն ու գրառման հաջորդականությունը: Ընդ որում՝ ֆունկցիայից վերադարձվող արժեքի տիպը չի հանդիսանում վերաբեռնավորման սկզբունքի տարբերիչ հանգամանք. այստեղ էականը միայն պարամետրերի քանակական և որակական տարբերությունն է:

Գրենք հետևյալ խնդրի լուծման ծրագիրը. *միևնույն `gumar` անունը կրող վերաբեռնավորված երկու ֆունկցիաների միջոցով հաշվել*

ա) a և b իրական

և

բ) x և y ամբողջ թվերի գումարը:

```
#include <iostream.h>
void gumar(double m, double n, double *s) {*s=m+n;} //1
int gumar(int k,int e) {return k+e;} //2
void main()
{
    double a,b,ss ;
    cin >> a >> b;
    gumar(a,b,&ss) ; cout <<ss <<endl; //3
    int x,y;
    cin >> x >> y;
    cout <<gumar(x,y) <<endl ; //4
}
```

Ծրագրի //1 տողում հայտարարվել է առաջին վերաբեռնավորված ֆունկցիան, //2 տողում՝ երկրորդը: //3 տողի `gumar(a,b,&ss)` գրառմամբ կանչվում է //1 տողում որոշված ֆունկցիան, իսկ //4 տողի համաձայն՝ //2 տողում սահմանված ֆունկցիան:

Եթե ֆունկցիաներում իրականացվող գործողությունները միատեսակ են՝ նպատակահարմար է դրանք վերաբեռնավորել մեկ այլ եղանակով, որն անվանում են **շաբլոնային**: Շաբլոնային վերաբեռնավորման դեպքում *ֆունկցիաների մարմինները համընկնում են*. մնում է վերնագիրը կազմավորել այնպես, որ վերաբեռնավորված ֆունկցիաներից յուրաքանչյուրի կանչի դեպքում կոմպիլյատորը միարժեքորեն իմանա, թե *պարամետրերի ինչպիսի տիպերին է տվյալ պահին հարմարեցնելու շաբլոնը*: Ընդ որում՝ կարելի է շաբլոնով տալ ինչպես ֆունկցիայից վերադարձվող արժեքի տիպը, այնպես էլ պարամետրերի տիպերը:

Ֆունկցիաների շաբլոնային վերաբեռնումն իրականացվում է ըսպ վերադարձվող արժեքների և ֆորմալ պարամետրերի տիպերի:

Ֆունկցիայի *շաբլոնի (ընդհանրի)* ընդհանուր տեսքը հետևյալն է.

template <շաբլոնային պարամետրերի ցուցակ> շաբլոնային ֆունկցիայի մարմին

որտեղ *template*-ը առանցքային բառ է, իսկ *< >*-ի մեջ առնված *շաբլոնային պարամետրերի ցուցակը* պարունակում է ստորակետերով անջատված մեկ կամ մի քանի տիպերի շաբլոնային անվանումներ, որոնցից յուրաքանչյուրի դիմաց պետք է գրված լինի *class* կամ *typename* առանցքային բառերից որևէ մեկը: Այստեղ ներառված պարամետրերը, վերջին հաշվով, **ֆորմալ տիպեր** են, որոնք արժեքներ են ստանում ֆունկցիայի կանչի պահին:

Օրինակ՝ շաբլոնային ֆունկցիայի վերնագիրը կարող է լինել հետևյալ տեսքի՝

```
template <class T, class C>
T max(T x, C y)
```

որը կնշանակի, որ ֆունկցիան *շաբլոնացված է երկու իրարից տարբեր C և T տիպերի համար*:

Գրենք հետևյալ խնդրի լուծման ծրագիրը. *որոշել a, b իրական, c, d ամբողջ և e, f սիմվոլային տիպի մեծություններից մեծագույնները՝ երկու պարամետրերից մեծագույնը որոշող շաբլոնով վերաբեռնավորված ֆունկցիայի կիրառմամբ*:

```
#include <iostream.h>
template <class T>
T max (T x, T y)
    {if (x>y) return x; else return y;} //1
void main()
    {double a,b;
    cin >> a >> b;
    cout <<max(a,b) <<endl; //2
    int c,d;
    cin >> c >> d;
    cout <<max(c,d) <<endl; //3
```

```

char e,f;
cin >> e >> f;
cout << max(e,f) << endl; //4
}

```

Երբ //2 տողում *max* ֆունկցիան կանչվում է *double* տիպի *a* և *b* փաստացի պարամետրերի համար, թարգմանիչը *max* ֆունկցիայում ամենուրեք *T* շաբլոնները ավտոմատ փոխարինում է *double* տիպով և այդպիսով ոչ միայն *x*, *y* փոփոխականներն են ստանում *double* տիպ, այլև ֆունկցիայից վերադարձվող արժեքը: Երբ //3 տողում երկրորդ անգամ է *max*-ը կանչվում՝ արդեն *int* տիպի *c* և *d* փաստացի պարամետրերի համար, այս դեպքում շաբլոնի *T*-ն փոխարինվում է *int* , //4-ում առկա կանչի դեպքում՝ *char* տիպերով:

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ Որպեսզի ֆունկցիային փոխանցված զանգվածը պաշտպանեք անցանկալի փոփոխման հնարավորությունից՝ այն որպես ֆորմալ պարամետր նկարագրելիս *const* կիրառեք:
- ◆ Ֆունկցիան վերաբեռնավորելիս պետք է զգույշ լինել այն դեպքերում, երբ դրանց նկարագրություններում լռության սկզբունքով արժեքովոդ ֆորմալ պարամետրեր կան:
- ◆ Ֆունկցիաները չեն կարող վերաբեռնավորվել ըստ վերադարձրած արժեքի տիպի:
- ◆ Ֆունկցիաների շաբլոնային վերաբեռնավորման դեպքում ընդգրկված շաբլոնների անունները պետք է իրարից տարբեր լինեն:
- ◆ Երկչափ զանգվածը որպես ֆորմալ պարամետր նկարագրելիս առաջին ինդեքսի չափը բաց են թողնում, իսկ երկրորդինը՝ նշում:



1. Ինչպե՞ս է զանգվածը փոխանցվում ֆունկցիային:
2. Ինչպե՞ս կարելի է ֆունկցիային զանգվածի տարր փոխանցել:
3. Ֆունկցիաները վերաբեռնավորելիս արդյոք էսկա՞ն է դրանցով իրականացվող ալգորիթմների նմանությունը:
4. Ճիշտ վերաբեռնման համար ի՞նչ սկզբունքներ պետք է պաշտպանվեն:
5. Ի՞նչ է շաբլոնային վերաբեռնավորումը, ե՞րբ իմաստ ունի այն կիրառել:
6. Ֆունկցիայում շաբլոնային տիպ նկարագրելիս ի՞նչ առանցքային բառեր են օգտագործում:

§ 2.16 ԿԱՌՈՒՑՎԱԾՔՆԵՐ

Մենք արդեն ծանոթ ենք բաղադրիչներ պարունակող տիպի՝ զանգվածի հետ, որը միևնույն տիպն ունեցող տարրերի հավաքածու է: Սակայն հաճախ գործ ենք ունենում այնպիսի մեծությունների հետ, որոնք բնութագրվում են տարբեր տիպերի բաղադրիչների համադրությամբ:

Օրինակ՝ գրադարանում գրքի վերաբերյալ տեղեկություններ պարունակող քարտը կարող է ներառել հետևյալ տեղեկատվական բաղադրիչները՝

- հեղինակը (սիմվոլային տող),
- գրքի վերնագիրը (սիմվոլային տող),
- հրատարակման տարեթիվը (ամբողջ թիվ),
- էջերի քանակը (ամբողջ թիվ):

Եթե փորձենք այս բաղադրիչներով մեկ ամբողջություն հանդիսացող քարտը ձևակերպել որպես զանգված՝ դժվար կլինի, քանի որ բաղադրիչները տարբեր տիպի են՝ տարբեր երկարությամբ սիմվոլային տողեր և թվեր: Նման տարբեր տիպի բաղադրիչներով մեծություններ առօրյա կյանքում հաճախ են հանդիպում (օրինակ՝ ցանկացած աշակերտ բնութագրվում է իր անունով, ազգանունով, հայրանունով, ծննդյան տարեթիվով, սեռով և այլն):

Տարաբնույթ տվյալները մեկ տիպում համախմբելու համար C++ լեզվում նախատեսված են *կառուցվածքները (սպրուկիդուրա)*: Մենք կփորձենք նախ հասկանալ կառուցվածքի C լեզվում ունեցած իմաստը:

Կառուցվածքը կարող է պարունակել ինչպես մեկ, այնպես էլ բազմաթիվ տարբեր տիպերի բաղադրիչներ, որոնք կոչվում են *կառուցվածքի տարրեր* կամ *դաշտեր*:

Կառուցվածք հայտարարելու ընդհանուր եղանակը հետևյալն է.

```
struct կառուցվածքի անվանում
{կառուցվածքի դաշտեր;};
```

այստեղ *կառուցվածքի անվանումը* ցանկացած իդենտիֆիկատոր է, իսկ *կառուցվածքի դաշտերն* իրարից ;-երով տարանջատված բաղադրիչների հայտարարություններ են: Կառուցվածքի դաշտերն ավարտող *չևավոր փակագծին անպայման հաջորդում է կեղ-սպրուկիդուրա* (;):

Փորձենք գրադարանային քարտում առկա ինֆորմացիան ներկայացնել կառուցվածքի միջոցով՝

```
struct card
{
    char hexinak[40];           //հեղինակ
    char vernagir[20];        //վերնագիր
    int tari;                 //հրատարակման տարեթիվ
    int ej;                   //էջերի քանակ
};
```

Այսպիսով, ստեղծեցինք նոր՝ *card* անունով տիպ, որի միջոցով, օրինակ՝ 1000 գիրք ներառող գրադարանի քարտադարանը կարելի է նկարագրել հետևյալ կերպ՝

```
card x[1000];
```

Կարելի է կառուցվածքային տիպի հայտարարությունը համակցել տվյալ տիպի փոփոխականների հայտարարման հետ: Օրինակ՝

```
struct grich
{
    int qanak;
    float gin;
} y, x[10], *p;
```

Ըստ այս հայտարարության՝ ստեղծվել է կառուցվածքային նոր՝ *grich* անվամբ տիպ ու հայտարարվել են այդ տիպի *y* փոփոխականը, տաս նման տիպի տարր պարունակվող *x* զանգվածն ու *p* ցուցիչը:

Կառուցվածքի յուրաքանչյուր բաղադրիչին (այլ խոսքով՝ տարրին կամ դաշտին) դիմելու համար հատուկ գրելաձև է կիրառվում՝

կառուցվածք տիպի փոփոխական . կառուցվածքի դաշտ:

օրինակ՝ *grich* տիպի *y* փոփոխականի *qanak* դաշտին կարելի է դիմել *y.qanak* գրառմամբ:

Լուծենք հետևյալ խնդիրը. *դասարանի 30 աշակերտներից յուրաքանչյուրի վերաբերյալ ունենք հետևյալ ինֆորմացիան՝*

- ա) դասամատյանում նրա համարը,*
- բ) անուն-ազգանունը,*
- գ) ինֆորմատիկա առարկայից ստացած տարեկան նիշը:*

Պահանջվում է արտածել դասարանում այդ առարկայից գերազանց ստացողների քանակն ու անուն-ազգանունները:

Գրենք համապատասխան ծրագիրը.

```
#include <iostream.h>
void main()
{
    struct ashakert //1
    {
        short hamar;
        char amun_azganun[20];
        short nish;
    } x[30]; //2
    int i, ger_qanak=0; //3
    for(i=0;i<30;i++) //4
    {
        cout << "Mutqagreq matyani hamar@";
        cin >> x[i].hamar;
```

```

cout << "Mutqagreq anun-azganun@";
cin >> x[i]. anun_azganun;
cout << "Mutqagreq gnahatakan@";
cin >> x[i]. nish;
if ((x[i]. nish==9) || (x[i]. nish==10)) //5
    {
        ger_qanak++; //6
        cout <<x[i]. anun_azganun <<endl;
    }
cout << "Informatikayic gerazanc gnahatakan en stacel" <<
        ger_qanak << "ashakert" << endl;
} }

```

Ծրագրի //1 մեկնաբանությամբ տողում հայտարարվել է *ashakert* կառուցվածքային նոր տիպն ու այդ տիպի 30 տարր պարունակող x միաչափ զանգվածը: Գերազանց ստացած աշակերտների պահանջվող քանակը հաշվելու նպատակով //3 տողում կատարվել է *ger_qanak=0* նախնական վերագրումը: //4 տողում ներառված ցիկլի մարմնում ներմուծվել են աշակերտների տվյալները: Միմիկային զանգվածի առանձնահատկություններից ելնելով ասենք, որ բերված ծրագիրն աշխատեցնելիս անհրաժեշտ է անուն-ազգանուն մուտքագրելիս ներմուծել իրար կից՝ առանց բացատանիչ կիրառելու: Ներմուծմանը զուգընթաց //5 տողում առկա պայմանի միջոցով փնտրվել են գերազանց ստացողներն ու հաշվարկվել է նրանց քանակը:

Ինչպես տեսաք, կարելի է նաև կառուցվածքային տիպի ցուցիչ ունենալ. օրինակ, եթե տրված է *ashakert y,*p*; հայտարարությունը և կատարված է $p=&y$; վերագրումը, ապա p -ն ցույց տալով հիշողությունում կառուցվածքային տիպի y փոփոխականի տեղաբաշխման առաջին հասցեի վրա՝ նաև «տեղյակ» է, թե y -ը կազմող բաղադրիչ դաշտերը միասին քանի՞ բայթ են կազմում (բերված օրինակի համաձայն՝ p -ն ցույց կտա $(2+20+2)=24$ բայթ ներկայացնող դաշտի վրա, եթե ընդունենք, որ *short* տիպը 2 բայթ է զբաղեցնում):

Ցուցի միջոցով կառուցվածքի դաշտերին կարելի է դիմել \rightarrow նշանի օգնությամբ, որտեղ մինուսի և մեծի նշանների միջև բացատանիչ չկա. օրինակ՝

```
cout << p -> hamar;
```

հրամանով կարտածվի y կառուցվածքի *hamar* դաշտի թվային արժեքը: Այսպիսով, $y.hamar$ և $p \rightarrow hamar$ արտահայտությունները համարժեք են:

Լուծենք հետևյալ խնդիրը: *Տրված է կառուցվածքային 100 տարր պարունակող x զանգված, որի տարրերը գրադարանում առկա գրքերի վերաբերյալ հետևյալ տեղեկատվությունն են պարունակում.*

- ա) հեղինակի ազգանունը,
- բ) գրքի վերնագիրը,
- գ) էջերի քանակը,
- դ) գինը:

Եթե որևէ գրքի էջերի քանակը տրված k ամբողջ թվից ավել է, անհրաժեշտ է տվյալ գրքի գինը ֆունկցիայի միջոցով ավելացնել 2 անգամ:

```

#include <iostream.h>
const int n=100;
struct girq { char anun[20]; //1
              char vern[20];
              short ej;
              short gin;
              }; //2
void nor_gin(girq &); //3
void main()
{
    short i,k; girq x[n];
    for (i=0;i<n;i++)
    {
        cin >> x[i]. anun;
        cin >> x[i]. vern;
        cin >> x[i]. ej;
        cin >> x[i]. gin;
    }
    cin >> k;
    for (i=0;i<n;i++) //4
    {
        cout <<i <<"-rd grqi hin gin@=" <<x[i]. gin; //44
        if(x[i]. ej>k) nor_gin(x[i]); //5
        cout <<"isk nor gin@=" <<x[i]. gin; //6
    }
}
void nor_gin(girq & kk) //7
{ kk.gin*= 2;} //8

```

Քանի որ *nor_gin* ֆունկցիան պետք է *girq* տիպի մեծության հետ աշխատի, ապա անհրաժեշտ է *girq* կառուցվածքի հայտարարությունը տեղադրել ֆունկցիայի նախատիպից առաջ (*//3* տող): *nor_gin* ֆունկցիայի նախատիպից երևում է, որ ֆունկցիան որպես ֆորմալ պարամետր ստանում է *girq* տիպի փոփոխականի հղումը: Հիշենք. եթե պարամետրը փոխանցվում է արժեքով՝ ֆունկցիայում դրա պատճենն է ստեղծվում: Այս դեպքում կառուցվածքի տարրը որպես արժեք փոխանցելիս պետք է պատճենվեին դրա բոլոր բաղադրիչները, ինչը ցանկալի չէ: Մինչդեռ հղմամբ փոխանցելու դեպքում այդ վտանգը չկա: Ծրագրի հիմնական մարմնում զանգվածի և *k* թվի ներմուծումից հետո *//4* տողով սկսվում է խնդրի լուծման գործընթացը: Այստեղ, եթե պարզվում է, որ *i*-րդ գրքի էջերի քանակը մեծ է տրված *k*-ից, կանչ է կատարվում *nor_gin* ֆունկցիային, որին ուղարկվում է *x* զանգվածի *girq* տիպի կառուցվածք ներկայացնող *i*-րդ տարրը: Քանի որ ֆունկցիայում ստեղծվում է փոխանցվող տարրի հղումը (երկրորդ անունը), ապա մնում է գրքի հին գնի զբաղեցրած հասցեում գրել նոր գինը, որն արվում է *//8* տողով:

Այսպիսով, ծրագրի *//44* տողով, մինչև ֆունկցիային դիմելը, արտածվում է հին գինը, իսկ *//6* տողով՝ նորը:

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ C++-ում կառուցվածքները որպես բաղադրիչ կարող են նաև ֆունկցիա պարունակել:
- ◆ Կառուցվածքները ֆունկցիային ավտոմատ կերպով փոխանցվում են որպես արժեք պարամետրեր:
- ◆ Որպեսզի կառուցվածքը ֆունկցիային փոխանցելիս ժամանակի կորուստը չունենաք՝ այն փոխանցեք հղմամբ:
- ◆ Եթե p -ն ցուցիչ է մի կառուցվածքի վրա, որն ունի, օրինակ, a բաղադրիչը, ապա $p \rightarrow a$ կ ($*p$). a արտահայտությունները համարժեք են, երկուսն էլ դիմում են կառուցվածքի a դաշտին, մինչդեռ $*p.a$ -ն սխալ է, քանի որ $.$ (կեպ) գործողությունն առավել բարձր կարգի գործողություն է, քան $*$ -ը:



1. Տարբեր տիպերի բաղադրիչների համախմբություն ներկայացնող մեծությունն ինչպե՞ս են հայտարարում:
2. Ինչպե՞ս կարելի է դիմել կառուցվածքի բաղկացուցիչ դաշտերին:
3. Ինչպե՞ս են ցուցիչ միջոցով դիմում կառուցվածքի դաշտերին:
4. Կարո՞ղ է կառուցվածքը հայտարարվել որպես լուրջ տիպ:
5. Կառուցվածքի տիպի փոփոխականը սովորաբար ինչպե՞ս է փոխանցվում ֆունկցիային:
6. Ժամանակի անհարկի կորուստը չունենալու համար ինչպե՞ս պիտի կառուցվածքը փոխանցվի ֆունկցիային:

§ 2.17 ԴԱՍԸ ՈՐՊԵՍ ԿԱՌՈՒՑՎԱԾՔ ՏԻՊԻ ԸՆԴՀԱՅՆՈՒՄ

C++-ի ունեցած ամենակարևոր տարբերություններից մեկը C լեզվից այն է, որ կառուցվածքային մեծություններն այստեղ ոչ միայն տվյալներ են ներառում, այլև ֆունկցիաներ: Չէ՞ որ առօրյա կյանքում ցանկացած խնդիր լուծելիս ելնում են խնդրի առանցք հանդիսացող օբյեկտի ոչ միայն *քանակական տվյալներից*, այլև դրա *ֆունկցիոնալ հնարավորություններից*: Այս առումով քանակական տվյալներն ու ֆունկցիոնալ հնարավորությունները մի ամբողջություն են կազմում և C++ լեզվում հատուկ ձևով միավորվելով՝ նոր տիպ կազմում, որն անվանում են **դաս**: Դասը *օբյեկտային կողմնորոշմամբ* լեզուների կարևորագույն գործիքն է: Դաս տիպին պատկանող փոփոխականները, ի տարբերություն սովորական փոփոխականների, կոչվում են *օբյեկտներ*:

Դասը օբյեկտը բնորոշող քանակական տվյալների ու հատկանշական ֆունկցիաների միավորումն է մի միասնական կառուցվածքի մեջ:

Դասը հայտարարում են հետևյալ կերպ.

```
class դասի_անուն
{
    դասի_քանակական_տվյալներ_և_ֆունկցիոնալ_բաղադրիչներ;
};
```

որտեղ *դասի_անունը* ցանկացած իդենտիֆիկատոր է (այն սովորական փոփոխականների անվանումներից տարբերելու համար երբեմն սկսում են մեծատառ *C* տառով), իսկ *դասի_քանակական_տվյալներն_ու_ֆունկցիոնալ_բաղադրիչները*՝ դասի տարրերը: *Քանակական_տվյալները* դասի օբյեկտը բնութագրող փոփոխականների հայտարարություններն են, իսկ *ֆունկցիոնալ_բաղադրիչները*՝ տվյալ օբյեկտին բնորոշ հնարավոր ֆունկցիաները:

Օրինակ՝ եթե սահմանենք ուղղանկյունների դասը, ապա քանակական տվյալները կարող են լինել ուղղանկյան կողմերը, իսկ ֆունկցիոնալ բաղադրիչներ՝ ուղղանկյան մակերեսը, պարագիծն ու անկյունագիծը հաշվող ֆունկցիաները:

Դասի քանակական տվյալները դասի անդամներն են, իսկ ֆունկցիաները՝ դասի մեթոդները:

Դաս սահմանելիս սովորաբար դասի անդամները խմբավորում են դասի, այսպես կոչված, *private*, իսկ մեթոդները՝ *public* բաժնում: *private* և *public* բաժինների տարբերությունն այն է, որ *private*-ի տակ հայտարարված մեծությունները դասից (դասի մեթոդներից) դուրս տեսանելի կամ, որ նույնն է, հասանելի չեն, իսկ *public* բաժնում ներառվածը հնարավոր է *տեսնել*՝ կիրառել դասից դուրս տվյալ *դասին_պատկանող_օբյեկտի միջոցով*: Բացի *public* և *private* բաժիններից, դասը կարող է պարունակել նաև *protected* բաժին, որին կժանոթանաք հետագայում: Դասի սահմանման մեջ կարող են իրար հաջորդող բազմաթիվ *public*, *protected* և *private* բաժիններ ներառվել. սովորաբար այս բաժիններից ցանկացածի տարածքն ավարտվում է, եթե հանդիպում է մյուս բաժինը բնորոշող *public*, *protected* կամ *private* առանցքային բառերից որևէ մեկը:

Դասի *private* բաժնում հայտարարված մեծություններն անվանում են *փակ*, իսկ *public* բաժնում հայտարարվածները՝ *բաց*:

Դասի ցանկացած մեթոդից դասի մնացած մեթոդներն ու անդամները հասանելի են՝ անկախ այն բանից, թե որ բաժնի (*public*, *private*) տակ են դրանք ներառված:

Օրինակ՝ *բնութագրենք ուղղանկյունների դասը հետևյալ կերպ.*

```
class C_uxxankjun
{
    private:
        int a_koxm;
```

```

    int b_koxm;
public:
    int makes( );
    int paragic( );
    ~C_uxxankjun( )
    C_uxxankjun( );
    C_uxxankjun(int,int);
};

```

Ինչպես տեսնում եք, բերված օրինակում դասն ունի միայն մեկական *private* և *public* բաժին: *private* բաժինը երկու փակ անդամ է պարունակում՝ ամբողջ տիպի *a_koxm* և *b_koxm* փոփոխականները, որոնք նախատեսված են ուղղանկյան կողմերի չափերը տալու համար: *public* բաժինը չորս մեթոդ է ներառում, որոնցից երկուսը՝ *makes()* և *paragic()* ֆունկցիաները նախատեսված են համապատասխանաբար ուղղանկյան մակերեսն ու պարագիծը հաշվելու համար, իսկ հաջորդ երեքը կրում են դասի *C_uxxankjun* անվանումը և դասի համար *հայրուկ ֆունկցիաներ* են հանդիսանում:

Կոնստրուկտոր

Կոնստրուկտորը դասի անվանումը կրող մեթոդ է: Եթե այն կա, ապա պարտադիր պետք է հայտարարված լինի *public* բաժնում: Կոնստրուկտորը կարող է պարամետրեր ստանալ, սակայն արժեք չի կարող վերադարձնել: Չնայած գիտենք, որ արժեք չվերադարձնող ֆունկցիաները բնորոշվում են *void* տիպով՝ կոնստրուկտորը ոչ մի ձևով, անգամ *void*-ով չի բնութագրվում: Ուղղանկյունների դասի բերված սահմանման մեջ *C_uxxankjun* անվամբ երկու ֆունկցիաներ կան: Կոնստրուկտորի ֆունկցիան կարելի է վերաբեռնավորել այնպես, ինչպես ցանկացած ֆունկցիա: Բերված օրինակում *C_uxxankjun* ֆունկցիան վերաբեռնավորված է՝ առաջին կոնստրուկտորը ֆորմալ պարամետրերի դատարկ ցուցակ ունի, մինչդեռ երկրորդը *int* տիպի երկու ֆորմալ պարամետրեր է պարունակում:

Ընդհանրապես կոնստրուկտորը հատուկ դեր ունի, *դասին պարկանող ցանկացած օբյեկտ դրա միջոցով է սրեղծվում* (կառուցվում կամ սկզբնարժեքավորվում): Կոնստրուկտորի ֆունկցիան ամեն անգամ կանչվում է *ավրոմայր*, երբ դասին պատկանող օբյեկտ է հայտարարվում: Դասին պատկանող օբյեկտ հայտարարելու համար անհրաժեշտ է դասից դուրս տալ դասի անունը և օբյեկտը. օրինակ՝ *C_uxxankjun ob*; արտահայտությամբ *ob* անունով *C_uxxankjun* դասի օբյեկտ հայտարարվեց. ըստ այս հայտարարման՝ ավտոմատ կաշխատի *C_uxxankjun ()* դատարկ պարամետրերով կոնստրուկտորը, իսկ *C_uxxankjun obb(5,7)*; հայտարարման դեպքում՝ պարամետրեր ընդունող *C_uxxankjun (int,int)* կոնստրուկտորը:

Եթե դասում կոնստրուկտոր չի հայտարարվում, լեզվի թարգմանիչը ավրոմայր կերպով դատարկ պարամետրերով ու դատարկ մարմնով կոնստրուկտոր է կցում դասին:

Եթե դասի օբյեկտը ստեղծելիս դիմամիկ հիշողություն է կիրառվում, ապա իմաստ ունի այդ հիշողությունն ազատել ամեն անգամ, երբ տվյալ օբյեկտը դառնում է «ավտոտրո» կամ անտեսանելի (ինչպես, օրինակ՝ ֆունկցիայում հայտարարված օբյեկտը, երբ ֆունկցիան ավարտում է աշխատանքը):

Դեստրուկտոր

Դասի դեստրուկտորը հատուկ ձևով կազմավորված ֆունկցիա է, որի աշխատանքի նպատակն օբյեկտի գրաված հիշողության ազատումն է: Այս ֆունկցիան աշխատում է ավտոմատ ամեն անգամ, երբ դասին պատկանող օբյեկտը հայտնվում է տեսանելիության տիրույթից դուրս: Դեստրուկտորը ոչ միայն կոնստրուկտորի նման արժեք չի վերադարձնում, այլև ոչ մի պարամետր չի ընդունում: Այսպիսով, այն հնարավոր չէ վերաբեռնավորել, միակն է կամ չկա: Դեստրուկտորի ֆունկցիան նույնպես կրում է դասի անունը, բայց կոնստրուկտորից տարբերվելու համար սկսվում է ~ (տիլդա) նշանով, օրինակ՝ `~C_uxxankjun()`-ը `C_uxxankjun` դասի դեստրուկտորի անվանումն է: Դեստրուկտորի ֆունկցիան նույնպես տիպ չունենալով՝ նաև `void` բառով չի կարող բնութագրվել:

Ասում են, որ կոնստրուկտորը **կառուցում** է օբյեկտը, իսկ դեստրուկտորը՝ **քանդում** կամ **ոչնչացնում**:

Դասը **սահմանել նշանակում է** `C_uxxankjun` դասի համար բերված օրինակի նմանությամբ **դասում փալ դրա մեթոդների նախադիպերը**: Այս դեպքում դասն ամբողջությամբ նկարագրելու համար պետք է դասից դուրս այդ մեթոդները նկարագրել: Դասի յուրաքանչյուր մեթոդ դրսում (դասից դուրս) նկարագրելիս դասի հետ կապ չունեցող ֆունկցիաներից տարբերելու համար պետք է նշել դասին դրա պատկանելու փաստը՝ արդեն հայտնի :: պատկանելության գործողությամբ: Օրինակ՝ `C_uxxankjun` դասի մակերեսը հաշվող ֆունկցիան դասից դուրս կարելի է նկարագրել հետևյալ կերպ.

```
int C_uxxankjun:: makes ( )
{return a_koxm * b_koxm ;}
```

Այստեղ, ինչպես նկատում եք, ֆունկցիայի `int` տիպը գրվել է վերնագրի սկզբում, որին հաջորդել է `C_uxxankjun` դասին `makes`-ի պատկանելու փաստի նշումը, ապա՝ ֆունկցիայի մարմինը:

Ընդհանրապես դասի կոնստրուկտորը, անկախ դրսից փոխանցվող արժեքներից՝ օբյեկտը պետք է ճիշտ կառուցի: Կոնստրուկտորի ֆունկցիան գրելիս պետք է ուշադիր լինել և այդ առումով միջոցներ ձեռնարկել: Այսպես, `C_uxxankjun` դասի պարամետրերով կոնստրուկտորն այդ առումով առավել «խոցելի» է, քանի որ փոխանցված որոշ (օրինակ՝ բացասական արժեքներով) պարամետրերի դեպքում «սխալ» կառուցված ուղղանկյուն կունենանք:

Գրենք ծրագիր, որն **օգտագործելով վերը սահմանված `C_uxxankjun` դասը՝ որոշի 7 ու 15 և ներմուծված `a` ու `b` կողմերով ուղղանկյունների մակերեսներն ու պարագծերը**:

```

#include <iostream.h>
class C_uxxankjun
{ private:
    int a_koxm;
    int b_koxm;
public:
    C_uxxankjun ();
    C_uxxankjun (int,int);
    ~C_uxxankjun ();
    int makeres ();
    int paragic ();
};

C_uxxankjun :: C_uxxankjun ()
//0
    { a_koxm = 7;
      b_koxm = 15;
    }

C_uxxankjun :: C_uxxankjun (int a1, int b1) //1
    { if (a1>0 && b1>0) {a_koxm = a1; b_koxm = b1;}
      else { if (a1<=0){a_koxm=7; b_koxm=b1;}
            if (b1<=0) {b_koxm=15; a_koxm=a1;}
          }
    }

C_uxxankjun :: ~C_uxxankjun() //2
    { cout << "ashxatec destruktork@" << endl; }

int C_uxxankjun :: makeres ()
    { return a_koxm * b_koxm;}

int C_uxxankjun :: paragic ()
    { return 2 * (a_koxm + b_koxm);}

void main ()
{   int x,y;
    cin >> x >> y;
    C_uxxankjun ob(x,y); //3
    cout << " ob uxxankjan makeres@" << ob.makeres() << endl; //5
    cout << " ob uxxankjan paragic@" << ob.paragic () << endl; //55
    C_uxxankjun obb ; //4
    cout << "7 li 15 koxmerov obb uxxankjan makeres@" <<

```

```

                                obb.makeres() << endl ; //44
cout << "obb uxxankjan paragic@= " << obb.paragic () << endl ;//45
C_uxxankjun ob1(-3,4) ; //6
cout << "7 և 4 koxmerov ob1 uxxankjan makeres@= " <<
ob1.makeres () << endl; //66
cout << "ob1 uxxankjan paragic@= " << ob1.paragic () << endl ;//67
}

```

Ինչպես նկատեցիք, դասի մեթոդներին դիմելիս (//5, //55, //44, //45, //66, //67 տողեր) օբյեկտի անվան (*ob*, *obb*, *ob1*) և կանչվող մեթոդի միջև դրվել է (.) գործողության նշանը՝ ինչպես կառուցվածքների բաղադրիչներին դիմելիս: Ըստ //3-րդ տողի հայտարարության՝ *C_uxxankjun* դասի *ob* օբյեկտ է ստեղծվել՝ *x* և *y* կողմերի համար ներմուծված արժեքներով: Քանի որ օբյեկտը ստեղծելիս *ob(x,y)*-ի համաձայն կանչվել է *C_uxxankjun (int,int)* նախատիպով կոնստրուկտորը, ապա կառուցվող ուղղանկյան կողմերը կախված են *x* և *y* փոփոխականների համար ներմուծված արժեքներից՝ եթե դրանք դրական են, ապա ըստ //1 կոնստրուկտորի կկառուցվի *x*, *y* կողմերով ուղղանկյուն, հակառակ դեպքում, եթե միայն *x*-ն է ոչ դրական՝ 7 ու *y*, իսկ միայն *y*-ի ոչ դրական լինելու դեպքում՝ *x* ու 15 կողմերով ուղղանկյուն կկառուցվի:

//4 տողում կառուցվող ուղղանկյան համար կկանչվի առանց պարամետրերի //0 կոնստրուկտորը, որը կողմերի արժեքները կսահմանի 7 և 15:

//6 տողում ևս մեկ օբյեկտ է կառուցվել՝ այս անգամ նորից կանչվել է //1 պարամետրերով կոնստրուկտորը. քանի որ որպես *a_koxm* ուղարկվել է -3 բացասական թիվը, ապա կոնստրուկտորը սխալն ուղղելով՝ *a_koxm*-ի համար կսահմանի 7 արժեք, իսկ *b_koxm*-ը կընդունի 4 արժեք:

Ծրագրի աշխատանքից հետո էկրանին, բացի արտածված մակերեսների և պարագծերի արժեքներից, կտեսնենք 3 անգամ իրար հաջորդող "ashxatec destruktork@" հաղորդագրությունը. սա բացատրվում է այն բանով, որ ստեղծված *C_uxxankjun* դասի երեք՝ *ob*, *obb* և *ob1* օբյեկտները ծրագրի ավարտին այլևս դուրս մնալով տեսանելիությունից՝ պետք է ոչնչացվեն. այդ նպատակով նախ պետք է ազատվեն այդ օբյեկտների կողմից զբաղեցված հիշողության տիրույթները: Այսպիսով, յուրաքանչյուր օբյեկտի համար ավտոմատ կանչվել և աշխատել է դասի դեստրուկտորը: Ընդ որում՝ ծրագրի կատարման ընթացքում ինչ հաջորդականությամբ որ օբյեկտները ստեղծվում են՝ դրան հակառակ հաջորդականությամբ էլ ոչնչանում են: Այսպիսով, նախ կկանչվի վերջում ստեղծված *ob1*-ի, ապա *obb*-ի, իսկ վերջում՝ առաջինը ստեղծված *ob* օբյեկտի դեստրուկտորը:

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ **Դասի անդամները հայտարարելիս դրանք չի կարելի սկզբնարժեքավորել:**
- ◆ **Դասի մեթոդները տեսանելի են միմյանց համար և մեկը մյուսին կանչելիս օբյեկտի անհրաժեշտություն չունեն:**
- ◆ **Դասի մեթոդներն ու անդամները դասից դուրս անտեսանելի են, եթե դասին պատկանող օբյեկտ չկա:**

- ◆ **Դասին պարկանոդ օբյեկտի միջոցով կարելի է դիմել միայն դասի public բաժնում ներառված ինֆորմացիային:**
- ◆ **Դաս և կառուցվածք հասկացությունները հիմնականում համարժեք են՝ մի փարբերությամբ, որ եթե դասի հայտարարության մեջ բաժնի անունը բաց թողնվի, ապա հայտարարված փվյալները կհամարվեն փակ՝ private, իսկ կառուցվածքի մեջ public է:**
- ◆ **Դասի դեպրոկտորն ազատելով հիշողությունն օբյեկտի զբաղեցրած տարածքից՝ օբյեկտը չի ոչնչացնում:**



1. Ի՞նչ է դասը և ինչպե՞ս են այն հայտարարում:
2. Ի՞նչն են անվանում՝ ա) դասի անդամ, բ) դասի մեթոդ:
3. Private բաժնում հիմնականում դասի ո՞ր բաղադրիչներն են փեղ գրվում:
4. Public բաժնում հիմնականում դասի ո՞ր բաղադրիչներն են գրվում:
5. Ո՞րն է դասի կոնստրուկտորի դերը:
6. Ի՞նչ է դեպրոկտորը, ինչի՞ համար է այն կիրառվում:
7. Կարելի՞ է գերբեռնել կոնստրուկտորը, իսկ դեպրոկտորը՞:
8. Մի քանի օբյեկտի առկայության դեպքում ի՞նչ հաջորդականությամբ է աշխատում դեպրոկտորը:

§ 2.18 ԺԱՌԱՆԳՎԱՆՑ

Ընդհանրապես բնության մեջ էվոլյուցիան հնարավոր է դառնում ժառանգման շնորհիվ, երբ նոր սերունդը, ժառանգելով իրեն ստեղծած սերնդի ձեռքբերումները, եղածին ավելացնում է սեփականը: Այսպիսով, էվոլյուցիայի վերջին օղակում հայտնված սերունդը՝ հենվելով նախնիների ձեռքբերումների վրա, դառնում է տվյալ շղթայի «ամենահարուստ» ներկայացուցիչը:

Ծրագրավորման ասպարեզում այս առումով դասը որակական նոր, հզոր միջոց ձեռք բերեց՝ **ժառանգման** մեխանիզմի մշակմամբ:

Ասում են, որ *B* դասը *ժառանգվել է A դասից*, եթե նկարագրվել է հետևյալ 3 հնարավոր եղանակներից որևէ մեկով՝

- ա) *class B: public A { };*
- բ) *class B: protected A { };*
- գ) *class B: private A { };*

Այս գործընթացում *A* դասը համարում են հենքային՝ **բազային դաս**, իսկ *B*-ն՝ **ժառանգ** կամ **ստանցված դաս**:

Ժառանգման ա) եղանակը համարում են **բաց**, բ) եղանակը՝ **պաշտպանված**, իսկ գ)-ն՝ **փակ ժառանգում**:

Մենք առավել մոտիկից կձանոթանանք առաջին՝ **ժառանգման բաց եղանակին**:

Ժառանգման բաց եղանակի դեպքում ասում են, որ ժառանգ դասի օբյեկտը միաժամանակ *հանդիսանում է նաև բազային դասի օբյեկտ, մինչդեռ հակառակը ճիշտ չէ՝* բազային դասի օբյեկտն իրենից ծնված ժառանգ դասի օբյեկտը չի հանդիսանում: Ժառանգման բաց եղանակի դեպքում ժառանգ դասի օբյեկտի միջոցով, բացի սեփական դասի բաց մեթոդներից, հասանելի են դառնում նաև բազային դասի բաց (*public*) միջոցները: Բազային դասի *protected* բաժնում հայտարարված միջոցները հասանելի են թե՛ տվյալ դասի և թե՛ ժառանգ դասի մեթոդների համար: Մինչդեռ «դրսից», նույնիսկ բազային կամ ժառանգ դասի օբյեկտների միջոցով, *protected* բաժինը մնում է անհասանելի՝ ինչպես *private* բաժինը:

Այսպիսով, *protected* բաժնի իմաստը միայն նրանում է, որ դրա տակ ներառվածը հասանելի դառնա ժառանգ դասի մեթոդների համար, սակայն *private*-ի նման դրսից մնա անհասանելի:

Ժառանգման մեխանիզմն այնպիսին է, որ ժառանգ դասի օբյեկտը նախ և առաջ օժտված է բազային դասի հնարավոր միջոցներով, ապա նաև սեփական դասի ընձեռած հնարավորություններով:

Ժառանգ դասի օբյեկտի ստեղծման համար նախ աշխատում են բազային, ապա ժառանգ դասի կոնստրուկտորները, որոնցից յուրաքանչյուրն օբյեկտի իր մասն է կառուցում (սա տրամաբանական է. եթե ծնողը գոյություն չունենա՝ «չի ծնվի» զավակը): Իսկ ժառանգ դասի օբյեկտի ոչնչացման գործընթացն իրականացվում է ճիշտ հակառակ ձևով՝ նախ ոչնչանում է օբյեկտի ժառանգ դասին պատկանող մասը, ապա՝ բազայինի հատվածը: Այսպիսով, ժառանգ դասի օբյեկտի ոչնչացման համար նախ կանչվում է ժառանգ դասի, ապա բազային դասի դեստրուկտորը:

Շարունակելով ուղղանկյունների դասի արդեն ծանոթ օրինակը՝ **այդ դասից ժառանգման միջոցով ստեղծենք քառակուսիների դաս**:

```
#include <iostream.h>
class C_uxxankjun
{
    protected : int a_koxm;
                int b_koxm;
    public : C_uxxankjun () { a_koxm = 5; b_koxm = 10; }
            C_uxxankjun (int a1, int b1)
            {
                if (a1>0 && b1>0) { a_koxm = a1; b_koxm = b1; }
                else { if (a1<=0) { a_koxm = 7; b_koxm=b1; }
                        if (b1<=0) { b_koxm = 14; a_koxm=a1; }
                }
            }
            int makes () { return a_koxm * b_koxm; }
            int paragic () { return 2 * (a_koxm + b_koxm); }
            ~C_uxxankjun () { cout << "Sa bazayini destructorn e" << endl; }
};
```



```

class C_qarakusi : public C_uxxankjun //0
{
private : char c;
public: ~C_qarakusi () { cout << "Sa jarangi destructorn e" << endl;}
        C_qarakusi (int, char);
        C_qarakusi ();
        void nkarel () ;

};

C_qarakusi :: C_qarakusi (int k, char s) : C_uxxankjun (k,k)
        {c=s ;}

C_qarakusi :: C_qarakusi () :
        C_uxxankjun (5,5)
        {c= '*' ;}

void C_qarakusi :: nkarel ()
        { int i,j ;
          for (i=1; i<=a_koxm; i++)
            {for (j=1; j<=b_koxm; j++)
              cout << c;
              cout << endl;
            }
        }

void main ()
{ int x,y; cin >> x >> y;
  C_uxxankjun ob(x,y) ; //1
  cout << ob. makeses () << endl;
  cout << ob. paragic () << endl;
  C_uxxankjun obb; //2
  cout << obb. makeses () << endl; //3
  C_qarakusi O(8, '_'); //4
  cout << O. makeses () << endl; //5
  cout << O. paragic () << endl; //6
  O.nkarel (); //7
  C_qarakusi O1; //8
  cout << O1. makeses () << endl;
  cout << O1. paragic () << endl;
  O1. nkarel (); //9
}

```

Այժմ ուսումնասիրենք գրվածը:

Նախ և առաջ նկատենք, որ *C_uxxankjun* դասի *private* անդամներն այժմ հայտարարվել են որպես *պաշտպանված` protected*: Պատճառն այն է, որ ժառանգ *C_qarakusi* դասում *nkarel()* մեթոդը պետք է հնարավորություն ունենա «տեսնելու» իր համար բազային հանդիսացող դասի անդամներին, մինչդեռ *private* բաժինը ժառանգ դասի համար մնում է փակ, անմատչելի: Քանի որ բազային դասում կիրառված ուրիշ

նոր տարրեր չկան (բացի նրանից, որ դասի մեթոդները ամբողջապես նկարագրվել են հենց դասի մարմնում), անցնենք *C_garakusi* դասի ուսումնասիրմանը:

Նախ //0 տողում բերված

```
class C_garakusi : public C_uxxankjun
```

հայտարարմամբ փաստվում է, որ *C_garakusi* դասը *C_uxxankjun* դասից ծնվել է բաց (*public*) ժառանգմամբ: *C_garakusi* դասը, բացի երկու կոնստրուկտորներից, պարունակում է նաև մեկ այլ մեթոդ՝ քառակուսի նկարելու համար նախատեսված *void nkarel ()* ֆունկցիան, որը էկրանին քառակուսի է նկարում՝ որպես միջոց օգտագործելով սեփական դասի միակ փակ անդամի՝ *c*-ի արժեքը:

Անսովոր տեսք ունեն *C_garakusi* դասի կոնստրուկտորները. նախ պարամետրով *C_garakusi (int k, char s)* կոնստրուկտորի վերնագիրն իրեն հաջորդող երկու կետից (:) հետո կանչել է *C_uxxankjun* դասի պարամետրով կոնստրուկտորին՝ *a_koxm* և *b_koxm* պարամետրերին փոխանցելով քառակուսու միակ կողմի՝ *k*-ի արժեքը. իսկապես, այս դեպքում ուղղանկյունների դասում ժառանգ դասի մասը ճիշտ կկառուցվի՝ որպես հավասարակողմ ուղղանկյունի, այսինքն՝ քառակուսի: Այնուհետև կոնստրուկտորի իրագործվող մարմնում ներառված միակ հրամանն արժեքավորում է սեփական դասի *c* փակ անդամը: Եթե ժառանգ դասի կոնստրուկտորն այս ձևով (բացահայտ) չկանչեր բազային դասի համապատասխան կոնստրուկտորին, ապա մեքենան ավտոմատ կկանչեր բազային դասի առանց պարամետրերի կոնստրուկտորին (իսկ սա կկառուցեր 5 և 10 կողմերով ուղղանկյունի և ոչ թե քառակուսի):

C_garakusi դասի առանց պարամետրերի կոնստրուկտորն իր հերթին նույնպես կանչում է բազային դասի պարամետրերով կոնստրուկտորին՝ նորից թույլ չտալով «սխալ» օբյեկտ՝ ուղղանկյունի կառուցելու, իսկ սեփական *c* փակ անդամին էլ տալիս է ‘*’ արժեքը:

main ()-ում //1 տողում ներմուծված *x* և *y* կողմերով ուղղանկյունի է կառուցվել. դրանում համոզվելու համար հերիք է վերլուծել *makeres ()* և *paragic ()* ֆունկցիաների վերադարձրած արժեքները: Այսինքն՝ անկախ այն բանից, որ *C_uxxankjun* դասից այլ դաս է ժառանգվել, այն մնում է ինքնուրույն, անկախ դաս, և դրա օբյեկտի համար ոչինչ չի փոխվել:

//2 տողում նորից մեկ այլ ուղղանկյունի է ստեղծվել՝ այս անգամ դասի առանց պարամետրերի կոնստրուկտորի օգնությամբ (5 և 10 կողմերով). սրանում կհամոզվեք՝ էկրանին տեսնելով ստեղծված ուղղանկյան մակերեսի արժեքը (50):

//4-րդ տողում ստեղծվել է 8-ին հավասար կողմով քառակուսի, որի ստեղծման նպատակով *C_garakusi* դասի պարամետրով կոնստրուկտորը՝ կանչելով *C_uxxankjun* դասի պարամետրով կոնստրուկտորին՝ դրան է փոխանցում 8 թիվը որպես թե՛ *a_koxm* և թե՛ որպես *b_koxm*, բացի դրանից, սեփական դասի *private* անդամին՝ *c*-ին էլ փոխանցում է ընդգծման *_* նշանը: //5 և //6 տողերում արտածվածը ցույց կտա, որ քառակուսին ճիշտ է կառուցվել, և //7-ում *nkarel ()* ֆունկցիայի կանչի արդյունքում նկարվածը 8x8 կողմերով քառակուսի կներկայացնի:

//8 տողում *C_garakusi* դասի *O1* օբյեկտը ստեղծվում է առանց պարամետրի կոնստրուկտորի միջոցով (5 կողմով), այնպես որ *nkarel()* ֆունկցիայի միջոցով էկրանին *-ների օգնությամբ 5x5 կողմերով քառակուսի կնկարվի:

Ծրագրի աշխատանքի ավարտին նախ կոչնչանա ծրագրի վերջում ստեղծված *OI* օբյեկտը, որի համար կաշխատեն *C_garakusi* դասի, ապա բազային դասի դեստրուկտորները, այնուհետև *O* օբյեկտի ոչնչացման համար՝ նորից նույն հաջորդականությամբ նույն դեստրուկտորները, իսկ վերջում *obb*-ի և *ob*-ի համար՝ *C_uxxankjun* դասի դեստրուկտորը՝ մեկական անգամ:

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ Միևնույն դասից կարող են բազմաթիվ դասեր ժառանգվել:
- ◆ Դասը կարող է փարբեր բազային դասերից փարբեր չենթով ժառանգվել:
- ◆ Ժառանգ դասն իր հերթին կարող է բազային հանդիսանալ այլ դասերի համար:



1. *C++*-ում ժառանգման քանի՞ եղանակ կա. ինչպե՞ս են հայտարարում բաց եղանակը:
2. Բաց ժառանգման դեպքում բազային դասի օբյեկտը հանդիսանո՞ւմ է արդյոք ժառանգ դասի օբյեկտ:
3. Ժառանգ դասի օբյեկտի միջոցով կարելի է դիմել՝
 - ա) միայն սեփական դասի բաց մեթոդներին,
 - բ) բազային դասի *public* և *protected* մեթոդներին ու անդամներին,
 - գ) բազային դասի բաց մեթոդներին ու սեփական դասի փակ անդամներին,
 - դ) բազային և սեփական դասի բաց մեթոդներին ու անդամներին,
 - ե) բազային դասի փակ անդամներին:
 Ընդերեք ճիշտ փարբերակը:
4. Ո՞րն է *protected* բաժնի իմաստը:
5. Ժառանգ դասի օբյեկտ ստեղծելիս *n*՞ր կոնստրուկտորներն են աշխատում և ի՞նչ հաջորդականությամբ:
6. Ժառանգ դասի օբյեկտը ոչնչացվելիս *n*՞ր դեստրուկտորներն են աշխատում և ի՞նչ հաջորդականությամբ:

§ 2.19

ՎԻՐՏՈՒԱԼ ՖՈՒՆԿՑԻԱՆԵՐ: ԲԱԶՄԱԶԵՎՈՒԹՅՈՒՆ (ՊՈԼԻՄՈՐՖԻԶՄ)

Եթե ուղղանկյունների դասում, բացի մակերես և պարագիծ հաշվող ֆունկցիաներից, մտցնենք նաև ուղղանկյուն նկարող ֆունկցիա և դրան տանք նույն *void nkarel()* վերնագիրը, ապա ժառանգ *C_garakusi* դասի որևէ, օրինակ, *Ol* օբյեկտի միջոցով *Ol.nkarel()* ֆունկցիայի կանչի արդյունքում կաշխատի *C_garakusi* դասի *void nkarel()* ֆունկցիան, իսկ *Ol.C_uxxankjun :: nkarel()*; կանչի արդյունքում կաշխատի ուղղանկյունների դասի *nkarel()* մեթոդը:

Ենթադրենք, այս պարագայում (երբ ուղղանկյունների դասում ևս ունենք *nkarel()* անվամբ ֆունկցիա) կատարել ենք հետևյալ հայտարարությունները.

*C_uxxankjun ob, *p;*

C_garakusi obb;

Բազային հանդիսացող *C_uxxankjun* դասի ցուցչի միջոցով կարող ենք ցույց տալ թե՛ սեփական դասի և թե՛ ժառանգված դասի օբյեկտների վրա: Եվ եթե կատարում ենք *p=&ob;* վերագրումը (այսպիսով, բազային դասի ցուցչի միջոցով ցույց տալով բազային դասի օբյեկտի վրա), ապա *p->nkarel()*; կանչի արդյունքում աշխատում է բազային դասի *nkarel()*-ը: Ընդ որում՝ *p=&obb;* վերագրումից հետո (երբ բազային դասի ցուցչով ցույց են տալիս ժառանգ դասի օբյեկտի վրա) կատարված *p->nkarel()*; կանչի արդյունքում նորից կաշխատի ուղղանկյունների (բազային) դասի *nkarel()*-ը:

Այժմ փոխենք իրավիճակը. ենթադրենք, ուղղանկյունների դասի *nkarel()* մեթոդը հայտարարված է *virtual void nkarel()*; վերնագրով, այլ խոսքով, որպես, այսպես կոչված, *վիրտուալ ֆունկցիա*:

Վիրտուալ ֆունկցիան դասի մեթոդ է, որն այնուհետև, եթե վերահայտարարվում է ժառանգ դասերում՝ կրկին հանդես է գալիս որպես վիրտուալ:

Այս դեպքում, եթե բազային դասի ցուցչի միջոցով ցույց տրվի ժառանգ դասի օբյեկտի վրա՝ *p=&obb;* և կատարվի *p->nkarel()*; կանչը, ապա կաշխատի հենց ժառանգ դասում հայտարարված մեթոդը, իսկ *p=&ob;* վերագրումից հետո (*ob*-ն բազայինի օբյեկտ է) այդ նույն *p->nkarel()*; կանչով էլ կաշխատի բազային դասի համանուն մեթոդը: Այսպիսով ստացվում է, որ միևնույն *p->nkarel()*; կանչի դեպքում, կախված այն բանից, թե *p*-ն ո՞ր դասի օբյեկտն է ցույց տալիս, կաշխատի հենց տվյալ դասում առկա վիրտուալին համանուն ֆունկցիան: Այս երևույթը կոչվում է **բազմաչևորություն** կամ **պոլիմորֆիզմ**:

Ժառանգմամբ կապված դասերի օբյեկտների համար վիրտուալ ֆունկցիայի շնորհիվ սպեղծվող այն հնարավորությունը, որը բույլատրում է բազային դասի ցուցչի միջոցով միևնույն վիրտուալ ֆունկցիայի կանչով տարբեր մեթոդներ իրականացնել, անվանում են բազմաշեղծություն կամ պոլիմորֆիզմ:

Այսպիսով, եթե բազային դասի ցուցիչը ցույց է տալիս ժառանգ դասի օբյեկտի վրա, և այդ ցուցչի միջոցով բազային դասում առկա վիրտուալ ֆունկցիայի կանչ է կատարվում, ապա կանչվում է տվյալ ժառանգ դասում վիրտուալին համանուն ֆունկցիան:

Երբ ցուցիչը ծրագրի կադրաման փուլում է «ընդհատ» կանչվող մեթոդին, ապա աղայիսի գործընթացն անվանում են դինամիկ կապակցում, ի տարբերություն ստատիկ կապակցման, որը փեղի է ունենում ծրագրի քարզմանման փուլում:

Թվում է, թե ժառանգ դասում բազային դասի վիրտուալ ֆունկցիային համանուն ֆունկցիայի առկայությունը նման է ֆունկցիաների վերաբեռնավորման գործընթացին, սակայն բոլորովին այդպես չէ՝ վիրտուալ ֆունկցիային համանուն մյուս ֆունկցիաները ժառանգ դասերում պետք է ունենան բացարձակ ամեն ինչում համընկնող նույն վերնագիրը: Եթե ժառանգ դասում վիրտուալին համանուն ֆունկցիան այլ վերնագիր ունենա, ապա այս ֆունկցիայի նկատմամբ բազմաձևության հատկությունը չի գործի, քանի որ այս դեպքում արդեն գործ կունենանք ֆունկցիայի պարզ վերաբեռնավորման հետ:

Բազմաձևության գործընթացին ծանոթանանք հետևյալ խնդրի միջոցով. *սահմանել բազային դաս, որի միակ վիրտուալ մեթոդը վերադարձնում է double տիպի a և b պարամետրերի գումարը, որտեղ a-ն և b-ն այդ դասի protected (պաշտպանված) անդամներն են:* Այդ դասից ժառանգված առաջին դասում բազայինում հայտարարված վիրտուալին համանուն ֆունկցիան վերադարձնում է a-ի և b-ի արտադրյալը, իսկ երկրորդ ժառանգ դասում վիրտուալին համանուն ֆունկցիան վերադարձնում է a-ի և b-ի տարբերությունը: Տրված x և y double տիպի պարամետրերի համար հաշվել $x+y$, $x*y$ և $x-y$ արտահայտությունների արժեքները՝ օգտվելով բազային դասի ցուցչից:

Բերենք խնդիրը լուծող ծրագիրը.

```
#include <iostream.h>
class C_Baz{
    protected:
```

```
//1
```

```

        double a;
        double b;
    public: C_Baz(double a1, double b1)
        {a=a1; b=b1;}
        ~C_Baz() {cout << "Sa C_Baz-i destructorn e" << endl;}
    virtual double f() {return a+b;} //0
        }; //2
class jarang1: public C_Baz{ //3
    public: jarang1(double k1, double k2):C_Baz(k1,k2){ }
        ~jarang1() {cout << "Sa jarang1-i destructorn e";}
        double f() {return a*b ;} //00
        }; //4
class jarang2: public C_Baz{ //5
    public: jarang2(double m1, double m2):C_Baz(m1,m2) { }
        ~jarang2() {cout << "Sa jarang2-i destructorn e" << endl;}
        double f() {return a-b ;} //000
        }; //6

void main()
{ double x; y;
  cin >> x >> y;
  C_Baz ob1(x,y) , *p; //11
  p=&ob1;
  cout << p->f(); //a+b
  jarang1 ob2(x,y);
  p=&ob2;
  cout << p->f(); //a*b
  jarang2 ob3(x,y);
  p=&ob3 ;
  cout << p->f(); //a-b
}

```

Ուսումնասիրենք գրված ծրագիրը: Ծրագրի //1-ից //2-րդ տողերում նկարագրված է բազային *C_Baz* դասը, որը, բացի կոնստրուկտորի (*C_Baz*) և դեստրուկտորի (*~C_Baz*) ֆունկցիաներից, պարունակում է նաև վիրտուալ հայտարարված *virtual double f()* ֆունկցիան: Սա վերադարձնում է *a+b*-ի արժեքը, որտեղ *a*-ն և *b*-ն դասի պաշտպանված անդամներ են: Այնուհետև //3-ից //4-րդ տողերում հայտարարվել է *C_Baz*-ից ժառանգված *jarang1* դասը, որը, կոնստրուկտորից բացի, պարունակում է վիրտուալին համանուն ֆունկցիա: Այն վերադարձնում է *a*b*-ի արժեքը (//00):

Ծրագրի //5-ից //6-րդ տողերում հայտարարվել է *C_Baz*-ից ժառանգված 2-րդ դասը՝ *jarang2*-ը, որտեղ վիրտուալին համանուն ֆունկցիան վերադարձնում է *a-b*-ի արժեքը (//000):

Այժմ հետևենք *main()*-ի աշխատանքին:

//11 տողում ստեղծվել է բազային դասի *ob1* օբյեկտը, ինչպես նաև հայտարարվել է բազային դասի տիպի **p* ցուցիչը: Այնուհետև *p* ցուցիչը ստացել է բազային դասի

ob1 օբյեկտի հասցեն և այդ պատճառով $//a+b$ տողում $p \rightarrow f()$ կանչով կիրառործվի C_Baz -ի վիրտուալ $f()$ ֆունկցիան, որը վերադարձնում է $a+b$ -ի, վերջին հաշվով $x+y$ -ի արժեքը: Այնուհետև ստեղծվել է *jarang1* դասի *ob2* օբյեկտը և $p = \&ob2$; վերագրմամբ բազային դասի տիպի p ցուցիչը ստացել է *ob2*-ի հասցեն: Այժմ միևնույն $p \rightarrow f()$ կանչով կաշխատի *jarang1*-ում վիրտուալին համանուն $f()$ -ը, որը կվերադարձնի պահանջվող $a*b$ -ի արժեքը: Վերջում ստեղծվում է *jarang2* դասի *ob3* օբյեկտը և $p = \&ob3$; վերագրմանն հաջորդող $p \rightarrow f()$ կանչով իրագործվում է 2-րդ ժառանգ դասի $f()$ մեթոդը, որը հաշվում է $a - b$ -ի արժեքը:

Այսպիսով, բազմաձևության շնորհիվ միևնույն $p \rightarrow f()$ կանչով տարբեր դասերում հայտարարված ֆունկցիաներ կիրառործվեն:

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ Եթե ֆունկցիան որևէ դասում հայտարարվել է որպես վիրտուալ, ապա այդ դասից ժառանգված դասերում այն մնում է վիրտուալ:
- ◆ Եթե ժառանգ դասում վիրտուալ ֆունկցիան չի վերահայտարարվում, ապա այդ դասը բազայինից ժառանգում է վիրտուալ ֆունկցիայի նկարագրությունը:
- ◆ Եթե բազային դասում հայտարարված վիրտուալ ֆունկցիայի մարմինը փոխարինվում է $=0$ արտահայտությամբ (օրինակ՝ *virtual void f()=0*), ապա այդպիսի բազային դասն անվանում են արստրակտ դաս, իսկ այդ վիրտուալ ֆունկցիան՝ մաքուր վիրտուալ ֆունկցիա:
- ◆ Վիրտուալ ֆունկցիան կարող է կանչվել այնպես, ինչպես դասի ցանկացած այլ մեթոդ՝ առանց ցուցչի, դասի օբյեկտի միջոցով:
- ◆ Գեադրուկտորը կարող է լինել վիրտուալ, իսկ կոնստրուկտորը՝ ոչ:



1. Ինչպե՞ս են հայտարարում վիրտուալ ֆունկցիան:
2. Ո՞րն է բազմաձևությունը:
3. Ի՞նչ ֆունկցիա է իրագործվում, եթե բազային դասի ցուցիչը պարունակում է ժառանգ դասի օբյեկտի հասցեն և կանչ է կատարվում բազայինում հայտարարված վիրտուալ ֆունկցիային:
4. Ո՞րն են անվանում դինամիկ կապակցում:

§ 2.20 ԲԱՐԵԿԱՄ ՖՈՒՆԿՑԻԱՆԵՐ: ԲԱՐԵԿԱՄ ԴԱՍԵՐ

Ինչպես գիտեք, բացի դասի մեթոդներից՝ դասից դուրս որևէ միջոց չկա, որը թույլատրի օգտվել դասի փակ (*private*) անդամներից: Դասերի ժառանգման գործընթացին ծանոթանալուց հետո կարելի է փաստել, որ ժառանգ դասից հնարավորություն կա դիմել նաև բազային *protected*, բայց ոչ *private* բաժնին: Սակայն երբեմն անհրաժեշտ է լինում դասի փակ և պաշտպանված անդամներին դիմելու այնպիսի ֆունկցիայից, որը դասի անդամ չի հանդիսանում: Նման հնարավորություն ընձեռում է դասի անդամ չհանդիսացող, այսպես կոչված, **բարեկամ ֆունկցիան**:

Բարեկամ ֆունկցիաները նկարագրվում են այնպես, ինչպես սովորական ֆունկցիաները: Որպեսզի նման ֆունկցիան «թույլտվություն» ունենա դասի միջոցներից օգտվելու, դասի սահմանման մեջ պետք է հատուկ ձևով նշվի, որ այդ ֆունկցիան դասին *բարեկամ* է հանդիսանում: Դրա համար դասի *private*, *protected* կամ *public* բաժիններից որևէ մեկում պետք է տալ այդ *ֆունկցիայի նախաարխար*՝ այն սկսելով *friend* առանցքային բառով, օրինակ՝

```
friend void ff();
```

Դասին անդամ հանդիսացող ֆունկցիաները կանչի պահին ավտոմատ ստանում են այն օբյեկտի հասցեն, որի համար աշխատելու են. բարեկամ ֆունկցիան դասի անդամ չլինելով՝ այդպիսի հնարավորությամբ օժտված չէ և այդ պատճառով պետք է տվյալ դասի օբյեկտն ունենա:

Բարեկամ ֆունկցիան կարող է միաժամանակ մի քանի դասերի համար բարեկամ հանդիսանալ:

Բարեկամ ֆունկցիայի աշխատանքին ծանոթանանք՝ գրելով հետևյալ խնդրի լուծման ծրագիրը. *տրված է կառուցվածք հետևյալ դաշտերով՝ աշակերտի*

- ա) անունը,
- բ) ազգանունը,
- գ) բոլոր քննությունների միջին գնահատականը:

Սահմանել դաս, որի փակ անդամները երկուսն են՝ տրված կառուցվածքի տիպի 30 տարր պարունակող զանգվածի ցուցիչը և զանգվածի տարրերի քանակը: Դասն ունի մեթոդ, որը վերադարձնում է բոլոր աշակերտների միջին գնահատականը: Դասին բարեկամ ֆունկցիայի միջոցով արտածել այն աշակերտների անուններն ու ազգանունները, ովքեր այդ միջինից բարձր նիշ են վաստակել:

```
#include <iostream.h>
struct ashakert {
```

```
    char anun[10];
    char azganun[15];
    double nish;
```

```
//1
```



```

};
class C_dasaran { private:
    ashakert *p;
    int n;
    friend void f(const C_dasaran &, double); //2
    public:
        C_dasaran(ashakert *p1, int n1) {p=p1; n=n1;}
        ~C_dasaran () {}
        double migin()
        {
            int i; double s=0;
            for (i=0; i<n; i++) s+=p[i].nish;
            return s/n;
        }
};
void f(const C_dasaran &ob, double mm) //3
{
    int i; //4
    for (i=0; i<ob.n; i++) if (ob.p[i].nish > mm)
        cout << ob.p[i].anun << " " << ob.p[i].azganun << endl;
}
void main()
{
    int i, n; ashakert x[30];
    do {cin >> n;} while (n<1 || n>30);
    for (i=0; i<n; i++) //5
    {
        cout << "nermuceq" << i << "-rd ashakerti anun@";
        cin >> x[i].anun;
        cout << "nermuceq" << i << "-rd ashakerti azganun@";
        cin >> x[i].azganun;
        cout << "nermuceq" << i << "-rd ashakerti bolor
                                                                    qnmutyunneri migin nish@";
        cin >> x[i].nish;
    }
    C_dasaran obb(x,n); double m=obb.migin(); //6
    cout << "dasaratum miginic barcr en stacel hetevyal
                                                                    ashakertner@" << endl;
    f(obb,m); //7
}

```

Այժմ ուսումնասիրենք բերված ծրագիրը: Նախ //1 տողից տեղադրվել է *ashakert* կառուցվածքի հայտարարությունը, որին հաջորդել է *C_dasaran* դասի նկարագրությունը: Դասի *private* բաժնում, բացի խնդրում պահանջվող մեծություններից, տրվել է նաև դասին *f* ֆունկցիայի բարեկամ լինելը փաստող հայտարարությունը (*//2*)՝ հետևյալ կերպ՝

```
friend void f(const C_dasaran &, double);
```

Սա նշանակում է, որ դասին բարեկամ է ընդունվում *void* տիպի *f* ֆունկցիան, որին

որպես պարամետր պետք է փոխանցվեն *C_dasaran* դասի օբյեկտի հղումն ու իրական տիպի թիվ: *const* առանցքային բառն այստեղ ցույց է տալիս, որ ֆունկցիային իրավունք չի վերապահվում օբյեկտը փոփոխելու (այդ վտանգը կա, քանի որ ֆունկցիային փոխանցվում է օբյեկտի հղումը): Քանի որ *f*-ը չի հանդիսանում դասի մեթոդ, ապա այն նկարագրվում է դասից դուրս, այնպես, ինչպես դասից անկախ ցանկացած այլ ֆունկցիա (//3): *f*-ը, որպես *C_dasaran* դասին բարեկամ ֆունկցիա, ստանալով այդ դասի *ob* օբյեկտը, իրավունք ունի ոչ միայն դրա բաց մեթոդներին դիմելու, այլև՝ փակ անդամներին: Դրանից ելնելով, օգտվելով փակ անդամներից (*ob.n*, *ob.p[i]*), փնտրվել ու արտածվել են *mm*-ից բարձր նիշ ստացողների անուններն ու ազգանունները: *main*()-ում *C_dasaran*-ին պատկանող օբյեկտ ստեղծելու նպատակով նախ ներմուծվել են մինչև 30 հոգի պարունակող դասարանի աշակերտների տվյալները (//5), ապա ստեղծվել է դասի *obb* օբյեկտ (//6), որի օգնությամբ հաշվարկվել է դասարանի միջին նիշը (*m*), որն էլ ուղարկվել է *f* մեթոդին (//7):

Եթե ֆունկցիան բարեկամ է հանդիսանում մեկից ավելի դասերի համար, ապա այն պետք է ստանա (ունենա) այդ դասերի օբյեկտները: Օրինակ՝

```
class C_A { . . . . .
    . . . . .
    friend void bb ( C_A a,
                   C_B b);
};
class C_B { . . . . .
    . . . . .
    friend void bb ( C_A a,
                   C_B b);
};
```

Ինչպես երևում է բերված օրինակից՝ բարեկամ ֆունկցիայի վերնագրում նշվել են թե՛ *C_A* և թե՛ *C_B* դասերին պատկանող օբյեկտներ. մինչդեռ *C_B*-ն հայտարարված լինելով ավելի ուշ, քան *C_A*-ն, *bb*-ի առաջին հայտարարման դեպքում խնդիր է առաջացնում. այդ պատճառով լեզվում հնարավորություն է տրվում ավելի ուշ նկարագրվող *C_B* դասի **նախնական հայտարարություն** կատարելով *C_A*-ից առաջ՝ հակասությունը վերացնել.

```
class C_B;
class C_A { . . . . .
    . . . . .
};
class C_B { . . . . .
    . . . . .
};
```

Դասերի միջև ևս հնարավոր է «բարեկամություն» ստեղծել:

Որպեսզի C_B դասն ընդունվի C_A դասին բարեկամ՝ անհրաժեշտ է C_A դասի սահմանման մեջ ցանկացած բաժնում տալ հետևյալ հայտարարությունը.

friend class C_B;

Այս դեպքում C_B դասի բոլոր մեթոդները (նույնիսկ կոնստրուկտորը), ավտոմատ կերպով դառնում են C_A դասին բարեկամ մեթոդներ:

Դասերի բարեկամությունը միակողմ բարեկամություն է և հայտարարվում է վերից վար (ավելի վաղ հայտարարված դասն ընդունում է իրենից հետո հայտարարված դասի՝ իրեն բարեկամ լինելը), իսկ հակառակը հնարավոր չէ:

Դասի բարեկամությունը չի ժառանգվում. վերը բերված օրինակը շարունակելով ասենք, որ եթե C_B -ն բազային դաս է jar -ի համար, ապա թեպետ C_B -ն C_A -ի բարեկամն է, սակայն իր ժառանգը (jar) C_A -ին բարեկամ չի հանդիսանում:

Բացի դրանից, **դասի բարեկամությունը չի փոխանցվում.** եթե C_B -ն C_A -ի ընկերն է, իսկ C_C -ն C_B -ի ընկերը, ապա սա չի նշանակում, որ C_C -ն նաև C_A -ի ընկերն է:

ՕԳՏԱԿԱՐ Է ԻՄԱՆԱԼ

- ◆ **Ֆունկցիաների բարեկամությունը չի ժառանգվում:** Եթե բազային հանդիսացող դասի համար $f()$ մեթոդը բարեկամ է, ապա այդ դասից ժառանգված դասերի համար այն բարեկամ չի հանդիսանում:
- ◆ **Դասին պարկանող մեթոդը կարող է բարեկամ հանդիսանալ այլ դասի համար:**
- ◆ **Ֆունկցիան չի կարող լինել այն դասի անդամը, որին բարեկամ է համարվում:**



1. Ինչպե՞ս են անվանում այն ֆունկցիան, որը, դասի անդամ չլինելով, կարող է դիմել դասի փակ և պաշտպանված անդամներին:
2. Ինչպե՞ս է դասն ընդունում ֆունկցիայի բարեկամության փաստը:
3. Բարեկամ ֆունկցիան կարո՞ղ է առանց դասի օբյեկտի դիմել դասի անդամներին:
4. Բարեկամ ֆունկցիան կարո՞ղ է մեկ այլ դասի մեթոդ հանդիսանալ:
5. Ե՞րբ է դասի նախնական հայտարարություն կատարվում, ինչպե՞ս:
6. Ինչպե՞ս է հայտարարվում դասերի բարեկամությունը:
7. Ժառանգվո՞ւմ է արդյոք դասերի բարեկամությունը:

ՀԱՎԵԼՎԱԾՆԵՐ

Հավելված 1

C++-ի ստանդարտային բառերը

<i>asm</i>	<i>else</i>	<i>new</i>	<i>this</i>
<i>auto</i>	<i>enum</i>	<i>operator</i>	<i>throw</i>
<i>bool</i>	<i>explicit</i>	<i>private</i>	<i>true</i>
<i>break</i>	<i>export</i>	<i>protected</i>	<i>try</i>
<i>case</i>	<i>extern</i>	<i>public</i>	<i>typedef</i>
<i>catch</i>	<i>false</i>	<i>register</i>	<i>typeid</i>
<i>char</i>	<i>float</i>	<i>reinterpret_cast</i>	<i>typename</i>
<i>class</i>	<i>for</i>	<i>return</i>	<i>union</i>
<i>const</i>	<i>friend</i>	<i>short</i>	<i>unsigned</i>
<i>const_cast</i>	<i>goto</i>	<i>signed</i>	<i>using</i>
<i>continue</i>	<i>if</i>	<i>sizeof</i>	<i>virtual</i>
<i>default</i>	<i>inline</i>	<i>static</i>	<i>void</i>
<i>delete</i>	<i>int</i>	<i>static_cast</i>	<i>volatile</i>
<i>do</i>	<i>long</i>	<i>struct</i>	<i>wchar_t</i>
<i>double</i>	<i>mutable</i>	<i>switch</i>	<i>while</i>
<i>dynamic_cast</i>	<i>namespace</i>	<i>template</i>	

Հավելված 2

Ներկառուցված մաթեմատիկական ֆունկցիաներ

Վերնագրա- յին ֆայլ	Ֆունկցիա	Արգումենտի տիպը	Արդյունքի տիպը	Գործողությունը
<i>math.h</i>	<i>abs(x)</i>	<i>int</i>	<i>int</i>	$ x $
<i>math.h</i>	<i>fabs(x)</i>	<i>double</i>	<i>double</i>	$ x $
<i>math.h</i>	<i>cos(x)</i>	<i>double</i>	<i>double</i>	$\cos x$
<i>math.h</i>	<i>sin(x)</i>	<i>double</i>	<i>double</i>	$\sin x$
<i>math.h</i>	<i>tan(x)</i>	<i>double</i>	<i>double</i>	$\operatorname{tg} x$
<i>math.h</i>	<i>asin(x)</i>	<i>double</i>	<i>double</i>	$\arcsin x$
<i>math.h</i>	<i>acos(x)</i>	<i>double</i>	<i>double</i>	$\arccos x$
<i>math.h</i>	<i>atan(x)</i>	<i>double</i>	<i>double</i>	$\operatorname{arctg} x$
<i>math.h</i>	<i>log(x)</i>	<i>double</i>	<i>double</i>	$\ln x$
<i>math.h</i>	<i>log10(x)</i>	<i>double</i>	<i>double</i>	$\lg x$
<i>math.h</i>	<i>sqrt(x)</i>	<i>double</i>	<i>double</i>	\sqrt{x} , $x \geq 0$
<i>math.h</i>	<i>exp(x)</i>	<i>double</i>	<i>double</i>	e^x
<i>math.h</i>	<i>pow(x,y)</i>	<i>double</i>	<i>double</i>	x^y , սիսալ արդյունք կլրա, եթե $\begin{cases} x = 0 \\ y \leq 0 \end{cases} \text{ կամ } \begin{cases} x < 0 \\ y - \text{ը ամբողջ չէ} \end{cases}$
<i>math.h</i>	<i>pow10(x)</i>	<i>int</i>	<i>double</i>	10^x
<i>math.h</i>	<i>ceil(x)</i>	<i>double</i>	<i>double</i>	հաշվում է x արգումենտից ոչ փոքր մոտակա ամբողջը
<i>math.h</i>	<i>floor(x)</i>	<i>double</i>	<i>double</i>	հաշվում է x արգումենտին չգերազանցող մոտակա ամբողջը
<i>math.h</i>	<i>fmod(x,y)</i>	<i>double</i>	<i>double</i>	հաշվում է x -ը y -ի վրա բաժանելուց սրացված մնացորդը
<i>math.h</i>	<i>modf(x,y)</i>	<i>double</i>	<i>double</i>	վերադարձնում է x իրական բովի կոորդրակային մասը, իսկ ամբողջ մասը պահվում է y -ի մեջ

Հավելված 3

Գծային ալգորիթմների ծրագրավորում

1. Հաշվել և արտածել տրված քառանիշ թվի թվանշանների արտադրյալը:
2. Տրված եռանիշ թվի մեջ տեղերով փոխել միավորների և տասնավորների թվանշանների տեղերը: Արտածել ստացված նոր եռանիշ թիվը:
3. Տրված քառանիշ թվի մեջ տեղերով փոխել միավորների և հազարավորների, տասնավորների և հարյուրավորների թվանշանների տեղերը: Արտածել ստացված նոր քառանիշ թիվը:
4. Տրված են A , B , C և D իրական տիպի փոփոխականները, որոնք իրարից տարբեր արժեքներ ունեն: Կատարել հետևյալ փոփոխությունները. B -ն թող ստանա A -ի արժեքը, C -ն B -ի, իսկ D -ն՝ C -ի արժեքը: Արտածել A , B , C և D փոփոխականների նոր արժեքները:
5. Տրված x և a իրական ցանկացած արժեքների համար հաշվել և արտածել y -ի արժեքը, եթե.

$$\text{ա) } y = (x + 1) (x^2 + 1)^2 \sin(x + 3);$$

$$\text{բ) } y = \frac{x}{x^2 + 2} + 2^x,$$

$$\text{գ) } y = \operatorname{ctg} \frac{x}{x^2 + 1} + x^2,$$

$$\text{դ) } y = \sqrt[3]{x + 2} + \frac{x + 2}{x^2 + 6},$$

$$\text{ե) } y = \frac{x + 3}{x^2 + 2} + (|x| + 1) + x^2)^2,$$

$$\text{զ) } y = \operatorname{tg} \left(\frac{3x + 4}{x^2 + 4} \right) + \sqrt[3]{(x + 3)^2},$$

$$\text{է) } y = (x^2 + 4)^7 + \sin(\cos(x + a)),$$

$$\text{ը) } y = \sqrt[4]{x^2 + \sqrt[3]{x}} + (|x| + 1)^{10},$$

$$\text{թ) } y = \sin(z + 1)^2 + x^6 + \frac{1}{z}, \text{ որտեղ } z = \sqrt[5]{\frac{x + 4}{x^2 + 1}},$$

$$\text{ժ) } y = \sin \left(\frac{\pi}{12} + z \right) + e^{z+4}, \text{ որտեղ } z = \sin^2(\cos(x + a) + 1):$$

Ճյուղավորված ալգորիթմների ծրագրավորում

1. Հաշվել և արտածել տրված ֆունկցիայի արժեքը.

$$\text{ա) } Y = \begin{cases} 5x^2 - 2x - 3, & \text{եթե } x > 2, \\ x, & \text{հակառակ դեպքում:} \end{cases} \quad \text{բ) } Y = \begin{cases} 1 + x^2, & \text{եթե } x > 3, \\ \cos x + x, & \text{եթե } x \leq -1, \\ 1, & \text{հակառակ դեպքում:} \end{cases}$$

2. Արտածել *YES*, եթե տրված թիվը պատկանում է $[-10; 0]$ կամ $[2; 20]$ միջակայքերին, հակառակ դեպքում՝ *NO* հաղորդագրությունը:

3. Հաշվել և արտածել տրված երեք իրարից տարբեր թվերից մեծի արժեքը:

4. Հաշվել և արտածել տրված երեք իրարից տարբեր թվերից փոքրի արժեքը:

5. Հաշվել և արտածել տրված չորս իրարից տարբեր թվերից մեծի արժեքը:

6. Հաշվել և արտածել տրված չորս իրարից տարբեր թվերից փոքրի արժեքը:

7. Արտածել *YES*, եթե տրված a, b, c կողմերով եռանկյունը հավասարակողմ է, հակառակ դեպքում՝ *NO* հաղորդագրությունը:

8. Արտածել *YES*, եթե տրված $(x; y)$ կոորդինատներով կետը պատկանում է կոորդինատային հարթության երկրորդ քառորդին, հակառակ դեպքում՝ *NO* հաղորդագրությունը:

9. Հաշվել և արտածել կոորդինատային այն քառորդի համարը, որին պատկանում է տրված $(x; y)$ կոորդինատներով կետը:

10. Տրված են երեք թվեր: Հաշվել և արտածել բացասական թվերի քանակը:

11. Տրված երեք թվերից փոքրը մեծացնել մյուս երկուսի գումարի չափով: Տպել ստացված թվերը:

12. Տրված են երեք թվեր: Թվերն արտածել ըստ աճման հաջորդականության:

13. Տրված են երեք թվեր: Թվերն արտածել ըստ նվազման հաջորդականության:

14. Տրված են չորս թվեր: Թվերն արտածել ըստ աճման հաջորդականության:

15. Տրված են չորս թվեր: Թվերն արտածել ըստ նվազման հաջորդականության:

16. Տրված են չորս թվեր: Արտածել *I*, եթե դրանցից գոնե մեկը կենտ է, հակառակ դեպքում՝ *2*:

17. Տրված են չորս թվեր: Հաշվել և արտածել դրականների քանակը:

18. Տրված են եռանկյան կողմերի x, y և z երկարությունները: Արտածել *I*, եթե եռանկյունը ուղղանկյուն է, հակառակ դեպքում՝ *2*:

19. Տրված է եռանիշ թիվ: Արտածել *YES*, եթե եռանիշ թվի միավորների թվանշանը հավասար է տասնավորների և հարյուրավորների թվանշանների գումարին, հակառակ դեպքում՝ *NO* հաղորդագրությունը:

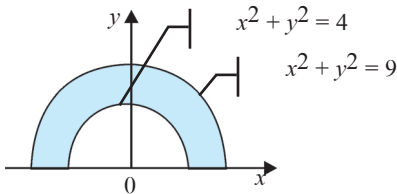
20. Արտածել *2*, եթե տրված թիվը երկնիշ է, *3*, եթե եռանիշ է, հակառակ դեպքում՝ *0*:

21. Տրված է եռանիշ թիվ: Արտածել *I*, եթե եռանիշ թվի թվանշանների գումարը գույգ է, հակառակ դեպքում՝ *0*:

22. Տրված է եռանիշ թիվ: Հաշվել և արտածել եռանիշ թվի թվանշաններից մեծագույնի արժեքը:

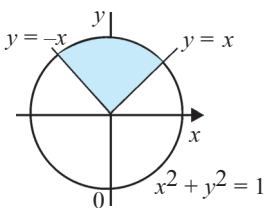
23. Տրված է եռանիշ թիվ: Հաշվել և արտածել եռանիշ թվի թվանշանների գումարի և եռանիշ թվի հարաբերության արժեքը, եթե միավորների թվանշանը մեծ է տասնավորների թվանշանից, հակառակ դեպքում կարտածի եռանիշ թիվը:
24. Տրված է քառանիշ թիվ: Արտածել 1 , եթե քառանիշ թվի թվանշանների մեջ կա 1 թվանշանը, հակառակ դեպքում՝ 0 թվանշանը:
25. Տրված է քառանիշ թիվ: Արտածել YES , եթե քառանիշ թվի միավորների և տասնավորների թվանշանների գումարը հավասար է 5 -ի, հակառակ դեպքում՝ NO հաղորդագրությունը:
26. Տրված է քառանիշ թիվ: Արտածել YES , եթե քառանիշ թիվը հավասար է իր թվանշանների գումարի քառակուսուն, հակառակ դեպքում՝ NO հաղորդագրությունը:
27. Տրված են կետի x և y կոորդինատները: Հաշվել և արտածել տրված ֆունկցիայի արժեքը, որտեղ $(x;y) \in D$ գրառումը նշանակում է, որ կետը պատկանում է նկարում ընդգծված D տիրույթին՝ եզրագծերով հանդերձ:

ա)



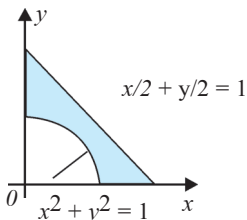
$$z = \begin{cases} 3x, & \text{եթե } (x;y) \in D, \\ 7x + 2y, & \text{հակառակ դեպքում:} \end{cases}$$

բ)



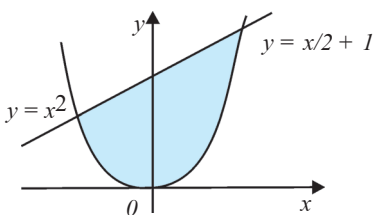
$$z = \begin{cases} \sin(x + y)^2, & \text{եթե } (x;y) \in D, \\ \cos x, & \text{հակառակ դեպքում:} \end{cases}$$

գ)



$$z = \begin{cases} x + y, & \text{եթե } (x;y) \in D, \\ x - y, & \text{հակառակ դեպքում:} \end{cases}$$

դ)



$$z = \begin{cases} \sin x, & \text{եթե } (x;y) \in D, \\ \cos x, & \text{հակառակ դեպքում:} \end{cases}$$

Կրկնության (ցիկլի) օպերատորներ

1. Որոշել $[1; 100]$ միջակայքի 3-ին բազմապատիկ տարրերի գումարը՝
 - ա) աճող պարամետրով ցիկլի կիրառմամբ,
 - բ) նվազող պարամետրով ցիկլի կիրառմամբ,
 - գ) նախապայմանով ցիկլի միջոցով,
 - դ) հետպայմանով ցիկլի միջոցով:
2. Որոշել և արտածել 12-ին բազմապատիկ ամենամեծ հնգանիշ թիվը:
3. Հաշվել և արտածել այն բնական թվերի քանակը, որոնց վրա առանց մնացորդի բաժանվում է տրված N բնական թիվը:
4. Հաշվել և արտածել տրված N թվից փոքր բոլոր բնական թվերի գումարը:
5. Արտածել այն հաջորդական 15 թվերը, որոնցից առաջինը հավասար է 2-ի, իսկ մնացածներից յուրաքանչյուրն իր նախորդից մեծ է 3 անգամ:
6. t տրամաբանական տիպի փոփոխականին վերագրել `true` արժեք, եթե տրված n բնական թիվը 3-ի աստիճան է, հակառակ դեպքում՝ `false`: Տպել t -ի արժեքը:
7. Տրված է n բնական թիվը: Ստանալ և տպել n -ից մեծ այն ամենափոքր թիվը, որը 2-ի աստիճան է հանդիսանում:
8. Տրված է N բնական թիվը: Հաշվել և արտածել N -ի կրկնակի ֆակտորիալը, որտեղ $N!! = N(N-2)(N-4)\dots$: Եթե N -ը գույգ է, ապա վերջին արտադրիչը հավասար է 2-ի, հակառակ դեպքում՝ 1-ի:
9. Տրված է N ($N > 1$) բնական թիվը: Հաշվել և արտածել Ֆիբոնաչիի թվերի հաջորդականությունը, որտեղ $F_1 = 1, F_2 = 1, F_k = F_{k-2} + F_{k-1}, k = 3, 4, \dots, N$:
10. Տրված է N բնական թիվը, որը 2-ի որևէ աստիճան է հանդիսանում՝ $N = 2^K$: Հաշվել և արտածել K -ի արժեքը:
11. Տրված է N բնական թիվը: Առանց քառակուսի արմատ հանելու ֆունկցիա կիրառելու հաշվել և արտածել այն ամենամեծ K բնական թիվը, որի քառակուսին չի գերազանցում N թվին՝ $K^2 \leq N$:

Միաչափ զանգվածներ

1. Հաշվել և արտածել տրված n տարր պարունակող միաչափ զանգվածի գույգ ինդեքս ունեցող տարրերի գումարը:
2. Հաշվել և արտածել տրված n տարր պարունակող միաչափ զանգվածի կենտ ինդեքս ունեցող տարրերի քառակուսիների արտադրյալը:
3. Հաշվել և արտածել տրված n տարր պարունակող միաչափ զանգվածի տրված $[a; b]$ միջակայքին պատկանող տարրերի գումարը:
4. Հաշվել և արտածել տրված n տարր պարունակող միաչափ զանգվածի այն տարրերի արտադրյալը, որոնք բացարձակ արժեքով փոքր են t թվից:
5. Հաշվել և արտածել տրված n տարր պարունակող միաչափ զանգվածի այն տարրերի միջին թվաբանականը, որոնց ինդեքսը բազմապատիկ է k ամբողջ թվին:

6. Հաշվել և արտածել տրված n ամբողջ տիպի տարր պարունակող միաչափ զանգվածի գույգ արժեք ունեցող տարրերի քանակը:
7. Հաշվել և արտածել տրված n տարր պարունակող միաչափ զանգվածի գրո արժեք ունեցող տարրերի քանակը:
8. Հաշվել և արտածել տրված n տարր պարունակող միաչափ զանգվածի m թվին բազմապատիկ տարրերի արտադրյալը:
9. Որոշել և արտածել տրված n տարր պարունակող միաչափ զանգվածի փոքրագույն տարրը:
10. Որոշել և արտածել տրված n տարր պարունակող միաչափ զանգվածի մեծագույն և փոքրագույն տարրերի գումարը:
11. Տրված n տարր պարունակող միաչափ զանգվածի դրական տարրերից նոր միաչափ զանգված ստանալ:
12. Տրված n տարր պարունակող միաչափ զանգվածի կենտ արժեք ունեցող տարրերից նոր միաչափ զանգված ստանալ:
13. Տրված n տարր պարունակող միաչափ զանգվածի $(c;d)$ միջակայքին պատկանող տարրերից նոր միաչափ զանգված ստանալ:

Երկչափ զանգվածներ

1. Հաշվել և արտածել 3×3 տարր պարունակող երկչափ զանգվածի կենտ արժեք ունեցող տարրերի արտադրյալը:
2. Հաշվել և արտածել 2×3 տարր պարունակող երկչափ զանգվածի տարրերի գույգ արժեք ունեցող գումարը:
3. Հաշվել և արտածել 3×3 տարր պարունակող երկչափ զանգվածի երկրորդ տողի տարրերի արտադրյալը:
4. Հաշվել և արտածել 4×4 տարր պարունակող երկչափ զանգվածի երրորդ սյան տարրերի գումարը:
5. Որոշել և արտածել 4×4 տարր պարունակող երկչափ զանգվածի մեծագույն տարրի արժեքը:
6. Որոշել և արտածել 3×3 տարր պարունակող երկչափ զանգվածի մեծագույն և փոքրագույն տարրերի գումարը:
7. Տրված են m ամբողջ թիվը և $m \times m$ տարր պարունակող երկչափ զանգված: Ստանալ և արտածել միաչափ զանգված, որի տարրերը ստացվում են տրված երկչափ զանգվածի այն տարրերից, որոնց քառակուսիներն ընկած են տրված $[a; b]$ միջակայքում:
8. Տրված են m ամբողջ թիվը և $m \times m$ տարր պարունակող երկչափ զանգված: Ստանալ և արտածել միաչափ զանգված, որի տարրերը ստացվում են տրված երկչափ զանգվածի գլխավոր անկյունագծի գրոյին հավասար տարր պարունակող տողից: Ենթադրվում է, որ գլխավոր անկյունագիծն ունի միայն մեկ գրոյին հավասար տարր:
9. Տրված են n և k ամբողջ թվերն ու $n \times n$ տարր պարունակող երկչափ զանգված: Հաշվել և արտածել k -րդ տողի մեծագույն տարրը:

10. Տրված են n և m ամբողջ թվերն $n \times n$ տարր պարունակող երկչափ զանգված: Հաշվել և արտածել m -րդ սյան փոքրագույն տարրը:

Ենթաժրագրեր

1. Տրված է $n \times n$ տարր պարունակող երկչափ զանգված: Չանգվածի տրված k թվից մեծ տարրերից ստանալ նոր զանգված: Չանգվածի ստացումը կազմակերպել պրոցեդուրայի միջոցով:
2. Տրված է n ամբողջ թիվը և n տարրեր պարունակող միաչափ զանգվածը: Տրված զանգվածի ոչ դրական տարրերից ստանալ նոր զանգված: Տրված զանգվածի տարրերի ներմուծումը կազմակերպել պրոցեդուրայի միջոցով:
3. Տրված է n ամբողջ թիվը և $n \times n$ տարրեր պարունակող երկչափ զանգվածը: Ստանալ և արտածել միաչափ զանգված, որի տարրերը տրված մատրիցի $[a; b]$ միջակայքին պատկանող տարրերն են: Ստացված զանգվածի տարրերի արտածումը կազմակերպել պրոցեդուրայի միջոցով:

Տողային փվյալների մշակում

1. Հաշվել և տպել տրված տողում առկա a պայմանանշանների քանակը:
2. Եթե տրված տողը աջից և ձախից կարդացվում է նույն կերպ, ապա տրամաբանական t փոփոխականին վերագրել *true* արժեքը, հակառակ դեպքում՝ *false*:
3. Տրված է տող, որի մեջ կա միայն 1 հատ x պայմանանշան: Հաշվել այդ պայմանանշանին հաջորդող 0 պայմանանշանների քանակը:
4. Տրված է տող, որի մեջ կան միայն 2 հատ z պայմանանշաններ: Հաշվել այն պայմանանշանների քանակը, որոնք գտնվում են այդ 2 պայմանանշանների միջև:
5. Տրված տողի ամեն մի a պայմանանշանից հետո ավելացնել c պայմանանշան:
6. Տրված տողից արտաքսելով a պայմանանշանները՝ ստանալ նոր տող:
7. Տրված տողի մեջ բոլոր x պայմանանշանները փոխարինել 2 հատ y պայմանանշաններով:
8. Տրված տողից հեռացնել առաջին v պայմանանշանին հաջորդող պայմանանշանները:

Գրառումներ

1. Տրված է n ամբողջ թիվը և n տարր պարունակող զանգվածը: Չանգվածի տարրերը գրառումներ են, որոնց համար բաղադրիչներ են տվյալ դասարանի աշակերտների՝ ա) անունը և ազգանունը, բ) մեկ առարկայի քննական գնահատականը: Տպել այն աշակերտների ցուցակը, որոնք ստացել են տրված թվից բարձր գնահատական:
2. Տրված է n ամբողջ թիվը և n տարր պարունակող զանգվածը: Չանգվածի տարրերը գրառումներ են, որոնց համար բաղադրիչներ են տվյալ դասարանի աշակերտների՝ ա) ազգանունը, բ) մեկ առարկայի քննական միավորը, գ) դասամատյանի համարները: Տպել դասարանի այն աշակերտների դասամատյանի համարներն ու ազգանունները, ովքեր ստացել են անբավարար գնահատական:

3. Տրված է n ամբողջ թիվը և n տարր պարունակող զանգվածը: Չանգվածի տարրերը գրառումներ են, որոնց համար բաղադրիչներ են տվյալ դասարանի աշակերտների՝ ա) ազգանունները, բ) անունները, գ) հայրանունները: Տպել այն աշակերտների ցուցակը (ազգանուն, անուն, հայրանուն), որոնց ազգանունը սկսվում է A տառով:
4. Տրված է n ամբողջ թիվը և n տարր պարունակող զանգվածը: Չանգվածի տարրերը գրառումներ են, որոնց համար բաղադրիչներ են գրադարակում առկա գրքերի՝ ա) հեղինակների ազգանունները, բ) էջերի քանակը: Տպել բոլոր այն գրքերի էջերի գումարային քանակը, որոնց հեղինակների ազգանունները սկսվում են A տառով:
5. Տրված է n ամբողջ թիվը և n տարր պարունակող զանգվածը: Չանգվածի տարրերը գրառումներ են, որոնց համար բաղադրիչներ են օրվա մեկ հեռուստաալիքի՝ ա) հաղորդումների վերնագրերը, բ) ժամը, գ) բուպեն: Տպել այն հաղորդումների վերնագրերը, որոնք սկսվում են ժամը 19 անց 30-ից հետո:
6. Տրված է n ամբողջ թիվը և n տարր պարունակող զանգվածը: Չանգվածի տարրերը գրառումներ են, որոնց համար բաղադրիչներ են ձայնասկավառակում ձայնագրված երգերի՝ ա) անունները, բ) տևողությունները, գ) հեղինակների ազգանունները: Տպել բոլոր այն երգերի անունները և հեղինակների ազգանունները, որոնք ունեն տրված k ամբողջ թվին հավասար տևողություն:

Ֆայլեր

1. C : կուտակիչի հիմնային կատալոգում ստեղծված է ամբողջ տիպի բաղադրիչներով $D1.DAT$ ֆայլը: Նույն ֆայլի մեջ կենտ արժեք ունեցող տարրերից հետո գրել այդ ֆայլի մեծագույն տարրի արժեքը:
2. C : կուտակիչի հիմնային կատալոգում ստեղծված է n ամբողջ տիպի բաղադրիչներ պարունակող $D1.DAT$ ֆայլը: Ֆայլից հեռացնել կենտ արժեք ունեցող տարրերը: Հեռացված տարրերը գրել C : կուտակիչի հիմնային կատալոգի $D2.DAT$ ֆայլում:
3. C : կուտակիչի հիմնային կատալոգում ստեղծված է n իրական տիպի բաղադրիչներ պարունակող $D1.DAT$ ֆայլը: Ստեղծել նոր $D2.DAT$ ֆայլը, որի սկզբում գրված լինի ֆայլի տրված $[a;b]$ միջակայքին պատկանող բաղադրիչները, իսկ վերջում՝ մնացածները:
4. C : կուտակիչի հիմնային կատալոգում ստեղծված է $2n+1$ հատ իրական տիպի բաղադրիչ պարունակող $D1.DAT$ ֆայլը: Փոխել ֆայլի մեծագույն (միակը) բաղադրիչի և կենտրոնի տարրի տեղերը:
5. C : կուտակիչի հիմնային կատալոգում ստեղծված է $D1.DAT$ ֆայլը: Ֆայլից հեռացնել առաջին փոքրագույն տարրին (ոչ միակը) հաջորդող տարրերը:
6. C : կուտակիչի հիմնային կատալոգում ստեղծված է $D1.DAT$ ֆայլը: Ստեղծել նոր ֆայլ, որի մեջ գրված լինեն տրված ֆայլի 3-ին բազմապատիկ բաղադրիչները:
7. C : կուտակիչի հիմնային կատալոգում ստեղծված է իրական տիպի բաղադրիչներ պարունակող $DD1.DAT$ ֆայլը: Ստեղծել նոր $DD2.DAT$ ֆայլ, որում գրված լինի տրված ֆայլի մեծագույն տարրին (միակը) հաջորդող տարրերը:
8. C : կուտակիչի հիմնային կատալոգում ստեղծված է ամբողջ տիպի բաղադրիչներով $D1.DAT$ ֆայլը: Ֆայլից հեռացնել այն տարրերը, որոնք հաջորդում են ֆայլի միակ 5-ին բազմապատիկ տարրին:

9. C : կուտակիչի հիմնային կատալոգում ստեղծված են n իրական տիպի բաղադրիչներ պարունակող $D1.DAT$ և $D2.DAT$ ֆայլերը: Ստեղծել նոր $D3.DAT$ ֆայլ, որի մեջ գրված լինեն տրված երկու ֆայլերի համապատասխան համարներով տարրերից մեծերը:
10. C : կուտակիչի հիմնային կատալոգի YY ենթակատալոգում ստեղծված է իրական տիպի բաղադրիչներ պարունակող $D1.DAT$ ֆայլը: Չևափոխել այդ ֆայլը՝ սկզբում գրելով ֆայլի տրված a թվից փոքր տարրերը, հետո հավասար տարրերը, իսկ վերջում՝ մեծ տարրերը:
11. C : կուտակիչի հիմնային կատալոգում ստեղծված են $D1.DAT$ և $D2.DAT$ ամբողջ տիպի բաղադրիչներով ֆայլերը: $D1.DAT$ ֆայլում գրել տրված ֆայլերի այն բաղադրիչները, որոնք մեծ են տրված b թվից, իսկ $D2.DAT$ ֆայլում՝ b թվից փոքր բաղադրիչները:
12. C : կուտակիչի հիմնային կատալոգի BP ենթակատալոգի $FILE$ ենթակատալոգում ստեղծված է իրական տիպի բաղադրիչներով $DD1.DAT$ ֆայլը: Փոխել ֆայլի մեծագույն և փոքրագույն բաղադրիչների տեղերը: Ենթադրվում է, որ ֆայլում կան միայն մեկ մեծագույն և մեկ փոքրագույն բաղադրիչներ:
13. C : կուտակիչի հիմնային կատալոգում՝ $D2.DAT$ ֆայլը: Ֆայլերի բաղադրիչները իրական տիպի թվեր են: $D1.DAT$ ֆայլից հեռացնել վերջին 5 բաղադրիչները և նրա տեղը գրել $D2.DAT$ ֆայլի առաջին 5 բաղադրիչները:
14. C : կուտակիչի հիմնային կատալոգում ստեղծված են $2n+1$ իրական տիպի բաղադրիչներ պարունակող $D1.DAT$ և $D2.DAT$ ֆայլերը: $D1.DAT$ ֆայլի կենտրոնի տարրի փոխարեն գրել $D2.DAT$ ֆայլի մեծագույն բաղադրիչի արժեքը:
15. C : կուտակիչի հիմնային կատալոգում ստեղծված է իրական տիպի բաղադրիչներով $D1.DAT$ ֆայլը: Տրված ֆայլի փոքրագույն տարրից հետո ավելացնել ֆայլի մեծագույն տարրի արժեքին հավասար բաղադրիչ:
16. C : կուտակիչի հիմնային կատալոգում ստեղծված է $2n+1$ հատ ամբողջ տիպի բաղադրիչներով $D1.DAT$ ֆայլը: Ֆայլից հեռացնել կենտրոնի 7 տարրերը և դրանց տեղը գրել մեկից մինչև 7 թվերը:

Քովանդակություն

Ներածություն	3
1. Ծրագրավորման Պասկալ լեզվի հիմունքները	5
§ 1.1. Ծրագրավորման Պասկալ լեզու: Լեզվի աշխատանքային միջավայրը	5
§ 1.2. Պասկալ լեզվի տարրերը	10
§ 1.3. Պասկալ ծրագրի կառուցվածքն ու հիմնական բաժինները	13
§ 1.4. Տվյալների պարզագույն տիպեր: Տվյալների գրանցման առանձնահատկությունները	16
§ 1.5. Ստանդարտ մաթեմատիկական ֆունկցիաներ և արտահայտություններ: Պատահական թվերի գեներացում	18
§ 1.6. Թվաբանական և տրամաբանական արտահայտություններ	22
§ 1.7. Մեկնաբանություններ: Վերագրման օպերատոր: Ներմուծում	25
§ 1.8. Արտածում: Արտածման ձևաչափ	28
§ 1.9. Գծային ալգորիթմների ծրագրավորում	30
§ 1.10. Ճյուղավորման գործընթացն ալգորիթմներում: Ճյուղավորման (պայմանի) օպերատոր	34
§ 1.11. Ընտրության օպերատոր	40
§ 1.12. Կրկնության (ցիկլի) օպերատորներ	42
§ 1.13. Միաչափ զանգվածներ	48
§ 1.14. Երկչափ զանգվածներ	51
§ 1.15. Ենթածրագիր-պրոցեդուրա	55
§ 1.16. Ենթածրագիր-ֆունկցիա	60
§ 1.17. Չանգվածը որպես ենթածրագրի պարամետր	64
§ 1.18. Տողային տիպի տվյալների մշակում	68
§ 1.19. Տողային փոփոխականների մշակման ստանդարտ պրոցեդուրաներ	72
§ 1.20. Գրառումներ	75
§ 1.21. Ֆայլեր	78
§ 1.22. Ֆայլերի աշխատանքը սպասարկող օժանդակ ենթածրագրեր	82
2. Ծրագրավորման C++ լեզվի հիմունքները	86
§ 2.1. C++ ծրագրի աշխատանքային միջավայրը	86
Լաբորատոր աշխատանք 2.1. C++ ֆայլի ստեղծում	91
§ 2.2. C++ լեզվի շարահյուսությունը: Ունար գործողություններ	95
§ 2.3. Բինար գործողություններ: Թվաբանական և տրամաբանական արտահայտություններ: Օպերատոր	102
§ 2.4. Ալգորիթմներ	109
§ 2.5. Ճյուղավորման գործընթաց: Ճյուղավորման (պայմանի) օպերատորներ: Անպայման անցման օպերատոր	113

§ 2.6. Կրկնության օպերատորներ: Break և Continue օպերատորներ	118
§ 2.7. Միաչափ զանգվածներ	124
§ 2.8. Երկչափ զանգվածներ	127
§ 2.9. Հղումներ: Ցուցիչներ	132
§ 2.10. Ջանգվածների հետ կապված ցուցիչներ	136
§ 2.11. Դինամիկ հիշողություն: Ջանգվածների տեղակայումը դինամիկ հիշողության տարածքում	139
§ 2.12. Հիշողության մեջ փոփոխականների բաշխման առանձնահատկությունները	141
§ 2.13. Ֆունկցիաներ: Ֆունկցիաների ցուցիչներ	144
§ 2.14. Ներկառուցվող (inline) ֆունկցիաներ: Ֆունկցիաներից արժեքներ վերադարձնելու այլ եղանակներ	149
§ 2.15. Ջանգվածների փոխանցումը ֆունկցիաներին: Ֆունկցիաների վերաբեռնավորումը	153
§ 2.16. Կառուցվածքներ	157
§ 2.17. Դասը որպես կառուցվածք տիպի ընդլայնում	161
§ 2.18. Ժառանգում	167
§ 2.19. Վիրտուալ ֆունկցիաներ: Բազմաձևություն (պոլիմորֆիզմ)	172
§ 2.20. Բարեկամ ֆունկցիաներ: Բարեկամ դասեր	176
Հավելվածներ	180
Հավելված 1	180
Հավելված 2	181
Հավելված 3	182

ԱՎԵՏԻՍՅԱՆ ՍԵՅՐԱՆ ՍԵՐԳԵՅԻ
ԴԱՆԻԵԼՅԱՆ ՍՎԵՏԻԿ ՎԱԶԳԵՆԻ

ԻՆՖՈՐՄԱՏԻԿԱ

11-րդ դասարան

ՎԵՐԱՀՐԱՏԱՐԱԿՈՒԹՅՈՒՆ

ՀԱՆՐԱԿՐԹԱԿԱՆ ԱՎԱԳ ԴԴՐՈՑԻ
ԲՆԱԳԻՏԱՄԱԹԵՄԱՏԻԿԱԿԱՆ ՀՈՍՔԻ ՀԱՄԱՐ

Խմբագիր՝ Արտակ Սուրենի Ոսկանյան

Սրբագրիչ՝	Անահիտ Պապյան
Ձևավորումը՝	Նվարդ Հայրապետյանի
Շապիկի ձևավորումը՝	Արամ Ուռուտյանի
Շարվածքը՝	Ալվարդ Ավետիսյանի

Պատվեր՝ 1230: Տպաքանակ՝ 5157:

Թուղթը՝ օֆսեթ: Չափսը՝ 70x100/16: 12 տպ. մամուլ:
Տառատեսակը՝ DallakTimeNew:

Տպագրված է «Տիգրան Մեծ» հրատարակչություն ՓԲԸ տպարանում