**Eksāmena jautājumi kursā "Algoritmi un programmēšana"**

**2024./2025. akadēmiskā gada rudens semestrī**

1. Algoritms. Tā uzdošanas veidi: cilvēku valoda, grafiska valoda, programmēšanas valoda.
2. Grafiskas valodas algoritma pierakstam. Pseidokods.
3. Kompilatori un interpretatori.
4. Programmēšanas vides un rīki.
5. Programmu testēšana un atkļūdošana.
6. Programmēšanas stils.
7. Datu izvade C++ konsoles režīmā.
8. Datu ievade C++ konsoles režīmā.
9. Datu ievade un izvade Python.
10. Programmēšanas valodas leksiskā līmeņa elementi.
11. Mainīgie (variables) un datu tipi (data types) valodā C++.
12. Mainīgie (variables) un datu tipi (data types) valodā Python.
13. Mainīgo deklarēšana, inicializācija un piešķiršana.
14. Veselo skaitļu datu tips (int) un loģiskais datu tips bool. Darbības un vērtību pieraksts.
15. Peldošā komata skaitļu datu tips (float, double). Darbības un vērtību pieraksts.
16. Veselo un peldošā komata skaitļu apstrādes īpatnības.
17. Mainīgo redzamība un eksistences periods. Lokālie un globālie mainīgie.
18. Skaitliskas izteiksmes (numeric expressions).
19. Loģiskas izteiksmes (logical expressions).
20. Skaitlisku vērtību apstrāde C++ un Python.
21. Zarošanās priekšraksts if-else. Vairāku līmeņu zarošanās.
22. Zarošanās priekšraksts switch un nosacījuma funktors (?:).
23. Cikla priekšraksti – for, while, do-while.
24. Izeja no cikla, apejot cikla nosacījumu – continue, break, return.
25. C++ programmas vispārējā struktūra (funkcijas, bloki, priekšraksti).
26. C++ programmas failu struktūra un kompilēšana.
27. Python programmas vispārējā struktūra.
28. Funkcija C++ – deklarācija, definīcija, izsaukums.
29. Funkcijas parametri un atgriežamais tips C++.
30. Funkcija Python.
31. Funkciju pārslogošana un noklusētie parametri.
32. Funkciju šablons C++ un parametru dinamisms Python.
33. Rekursija.
34. Masīvi C++ un to nodošana caur parametru.
35. Masīvi Python.
36. Vairākdimensiju masīvi C++ un to nodošana caur parametru.
37. Dinamiski masīvi C++.
38. Iebūvētais dinamiskais masīvs STL::vector valodā C++.
39. Simbola datu tips C++ (char). Darbības un vērtību pieraksts.
40. Zema līmeņa simbolu virkne C++ char\*. Darbības.
41. Augsta līmeņa simbolu virkne C++ string. Darbības.
42. Simbolu virkne Python.
43. Klases vispārējā struktūra C++. Metožu realizācija ārpus klases.
44. C++ konstruktors. Konstruktoru specifiskā sintakse.
45. Noklusētais konstruktors. Kopijas konstruktors.
46. C++ destruktors. Noklusētais destruktors.
47. Klases elementu redzamība. Klases statiskie un lokālie mainīgie.
48. Rādītājs, adrese un reference.
49. Formālie programmēšanas valodas sintakses definēšanas līdzekļi.
50. UML klašu diagrammas.

# Algoritms. Tā uzdošanas veidi: cilvēku valoda, grafiska valoda, programmēšanas valoda

**Algoritms ir precīzu, viennozīmīgu instrukciju virkne, kuru izpildes rezultātā tiek atrisināts noteikts uzdevums.**

Algoritmam ir noteikta izeja un ieeja. Jebkura algoritma pieraksts sastāv no divu veidu informācijas:

* **izpildāmās instrukcijas** (etapi),
* **vadības konstrukcijas**, kas nosaka instrukciju secību vienai aiz otras.

Trīs galvenās vadības konstrukcijas:

* 1. **Secība** – Nosaka vienas instrukcijas izpildi tieši aiz otras instrukcijas
  2. **Zarošanās** – Atkarībā no noteikta kritērija izpildīšanās nosaka, pa kuru no zariem turpināsies algoritma izpilde
  3. **Cikls** – Kādu instrukciju kopuma vairākkārtēja izpilde

Algoritmu uzdošanas veidi:

**Cilvēku valoda**

Algoritms cilvēku valodā ir uzdots vārdiskā pierakstā. Piemēram, pamācība, kā kāpināt skaitli *n* *k*-tajā pakāpē, kur *k* ir naturāls skaitlis.

1. solis: par rezultātu ņem skaitli 1.
2. solis: ja k ir lielāks par 0,

tad sareizināt rezultātu ar n, samazināt k par 1 un doties uz 2. soli.

pretējā gadījumā doties uz 3. soli.

1. solis: algoritms apstājas. Rezultāts glabā skaitli n k-tajā pakāpē.

**Grafiska valoda**

Piemēram, blokshēma, Nassi-Šneidermana struktogramma un UML diagramma. Grafiskā valodā konstrukcijas un instrukcijas attēlo specifiskā veidā, piemēram, blokshēmā darbībai atbilst taisnstūris, sazarošanai atbilst rombs u. tml.

Blokshēmai un UML noteikti jābūt sākumam un beigām kā atsevišķai instrukcijai, un tām nav cikls kā neatkarīga konstrukcija – to nodrošina sazarošanās ar atgriezenisko saiti. Nassi-Šneidermana struktogrammai ir cikls kā neatkarīga konstrukcija, bet tai ir mīnuss – ja kaut ko vajag izmainīt, tad nākas pārzīmēt daudz vairāk nekā blokshēmā un UML.

**Programmēšanas valoda:**

Programmēšanas valoda ir algoritma pieraksta forma, kas nodrošina automātisku algoritma izpildi datorā vai arī ir ērti pārveidojama uz kādu datorā automātiski izpildāmu formu. Programmēšanas valodā sarakstīta programma ir konstrukciju kopums, kas apraksta noteiktu algoritmu.

# Grafiskas valodas algoritma pierakstam. Pseidokods

**Pseidokods ir pusformāls algoritmu pieraksta veids, kas atgādina pierakstu kādā programmēšanas valodā.**

Pseidokods, atšķirībā no programmēšanas valodām, dod iespēju pierakstīt vairākas idejas, darbības un konstrukcijas neformāli, tādējādi nodrošinot, ka algoritms tiek pierakstīts īsāk un lasītājam vieglāk uztveramā formā, kā arī samazinās piesaiste kādai konkrētai programmēšanas valodai. Aritmētisku un citu darbību pieraksts notiek, izmantojot matemātisko notāciju, bet piešķiršanas operators bieži vien ir bultiņas formā no vērtības, kuru piešķir, uz mainīgo, kurā ieraksta.

N\_K-TAJĀ\_PAKĀPĒ (N,K)

REZ ← 1

WHILE K >= 1 DO

REZ ← REZ \* N

K ← K – 1

END WHILE

RETURN REZ

END FUNCTION

Ir pseidokoda pieraksta sintakse, kurā netiek lietoti bloku noslēdzošie apzīmējumi END, tādējādi visu programmas strukturēšanu uzliekot atkāpju izmantošanai pirms komandām.

# Kompilatori un interpretatori

**Kompilators ir datorprogramma, kas pārveido noteiktā programmēšanas valodā uzrakstītu programmu par izpildāmu moduli (vispārīgā gadījumā – par programmu mašīnkodā, tādu, kuru dators var tiešā veidā izpildīt).**

C++ ir sistēmas valoda, kuras pirmkodu kompilators pārvērš mašīnkodā, kuru var palaist.

**1. fāze. Kompilēšana.** Katram C++ failam tiek uzbūvēts atbilstošs objektu fails (faila paplašinājums parasti obj), kopumā veidojot objektu kodu;

**2. fāze. Linkošana.** No iegūtā objektu koda, kā arī citu programmu, t.sk. standarta bibliotēku objektu koda tiek izveidots gala produkts – izpildāmais kods (izpildāmais modulis vai moduļi).

Pilnu izpildāmā koda iegūšanas procesu mēdz saukt arī par **uzbūvēšanu** (*build*), tomēr sarunvalodā tiek lietots termins kompilēšana.

**Interpretators nodrošina kādā programmēšanas valodā pierakstītas programmas izpildi tiešā veidā, lai nebūtu jāaizveido programmas kopija mašīnkodā, kuru pildīt vēlāk.**

Python ir skriptu valoda, kuras pirmkoda izpildi interpretators nodrošina tiešā veidā.

# Programmēšanas vides un rīki

**Programmas failu organizāciju, kā arī pareizu kompilatora un citu palīgprogrammu izsaukšanas secību izpildāmā koda iegūšanai, parasti nodrošina izstrādes programmēšanas vide** (piemēram, VS Code, Wing, Code::Blocks),

Programmas kompilēšana, izmantojot noteiktu izstrādes vidi, parasti iespējama vismaz divos variantos: triviālajā (bez projekta veidošanas) un parastajā (ar projekta veidošanu). Jebkura izstrādes vide, kompilējot programmu, vienu vai vairākas reizes izsauc kompilatoru (piemēram, izstrādes vide Code::Blocks pēc noklusēšanas izmanto kompilatorus gcc un g++).

Izmantojot programmēšanas vidi, programmētājs var abstrahēties no zema līmeņa kompilēšanas vai interpretēšanas detaļām un “ar vienas pogas spiedienu” nonākt no programmas līdz rezultātam. Tas ir iespējams, jo programmatūras izstrādes vide automātiski lieto programmēšanas rīkus.

**Rīki ir kompilators, kas sakompilē programmu, interpretators un atkļūdotājs, kas ļauj iet cauri programmai pa rindiņai, lai saprastu, kur ir kļūda.**

# Programmu testēšana un atkļūdošana

**Testēšana ir programmas pārbaudīšana, vai tā dara to, ko sagaida, ka tā darīs.**

Tā var pārbaudīt izvaddatu vai padarītā darba atbilstību kaut kādiem ievaddatiem.

**Atkļūdošana ir kļūdu atrašana un izlabošana programmā, ko bieži atrod ar testēšanu.**

# Programmēšanas stils

Rakstot programmu, ir svarīgi, lai tā strādātu pareizi un vēlams arī efektīvi, tomēr arī programmas pierakstīšanas stilam ir ļoti liela nozīme programmēšanā – programmētājs ir cilvēks, un tieši viņam ir jāskatās uz programmu, jāsaprot, kas programmā ir uzrakstīts, jāveic labojumi, un cilvēka psiholoģiskās īpašības nosaka to, ka informācijas pasniegšanas formai un veidam ir būtiska nozīme. Pareizā formā vai veiksmīgā stilā uzrakstīta programma ne tikai palīdz nākamajā dienā saprast, kas programmā ir bijis domāts, bet arī sekmē pašu domāšanas procesu, nodrošinot ne tikai kvalitatīvāku, bet arī ātrāku programmatūras izstrādes procesu.

**Programmēšanas stils ir programmas teksta veidošanas paņēmienu kopums, kas nodrošina tādas programmas iegūšanu, kas ir viegli lasāma, saprotama un modificējama.**

Programmēšanas stils izpaužas gan kā noteiktu likumu ievērošana, gan kā attieksme pret programmas koda veidošanu. Stils ir atkarīgs ne tikai no programmētāja, bet lielā mērā ietekmējas no programmēšanas valodas. Labai programmēšanas valodai no stila viedokļa piemīt šādas īpašības:

* **Skaidrība un ortogonalitāte** (*clarity and orthogonality*). Konstrukcijas ir skaidri saprotamas visās situācijās un nedod negaidītus efektus, tās kombinējot vienu ar otru.
* **Laba rakstāmība** (*writability*). Laba notācija palīdz rakstīt programmu.
* **Laba lasāmība** (*readability*). Programmas jēgu var viegli saprast bez papildus dokumentācijas.
* **Viennozīmīgums** (*no tricks*). Grūti izmantot konstrukcijas nepareizā nozīmē.

1. Vispārzināmie programmēšanas stila noteikumi

* **Izmantojiet programmas prologu, lai dokumentētu svarīgākās īpašības:**
  + nosaukums,
  + autors,
  + pirmās un pēdējās kompilācijas datums,
  + attīstības vēsture (datums, autors, izmaiņas, mērķis, komentārs), versiju nozīme
* Funkciju apraksti,
* Izsaucamās apakšprogrammas.
* Izmantojiet komentārus.
* Ierobežojiet atsevišķu programmas bloku izmērus (piemēram, 1 lappuse).
* Izvairieties no goto.

1. Atkāpes un bloki

* **Atkāpes jāizmanto tā, lai tās norādītu uz programmas elementu hierarhisko struktūru.**
* Iekļaut figūriekavās arī blokus no viena priekšraksta, lai izvairītos no potenciālām kļūdām, modificējot programmu.
* Figūriekavu blokus veidot tā, lai pēc bloka sākuma varētu viegli noteikt bloka beigas – vislabākais, lai aizverošā figūriekava būtu tieši zem attiecīgās atverošās. Piemēram, atverošā figūriekava tieši aiz galvas, pakārtotās instrukcijas ar vienu atkāpi, aizverošā figūriekava tieši zem galvas.

konstrukcijas galva {

instrukcija

....

}

1. Identifikatoru darināšana

* **Nosakiet un konsekventi lietojiet noteiktu stilu dažādu elementu vārdu darināšanā.**

variable\_name

functionName

* Mainīgo, funkciju, klašu un struktūru vārdus veidojiet saturīgus, lai pēc tiem būtu iespējams noteikt attiecīgā elementa nozīmi programmā.
* Izņēmums varētu būt vienīgi mainīgie, kas ir skaitītāji – tiem var atvēlēt arī pavisam īsus nosaukumus, piemēram, i, j, k.

1. Izteiksmju veidošana

* Izmantojiet iebūvētās funkcijas.
* Ļaujiet mašīnai darīt melno darbu.
* Nomainiet atkārtotas izteiksmes ar funkciju izsaukumiem.
* Izmantojiet iekavas, lai izvairītos no divdomīgām situācijām.

1. Programmas struktūra un vadības sistēma

* **Rakstiet vispirms viegli saprotamā pseidokodā (piemēram, grafiskā valodā), pēc tam pārveidojiet vajadzīgajā programmēšanas valodā.**
* Rakstiet programmu tā, lai to varētu lasīt no augšas uz apakšu.
* Izvairieties lietot iekļautus if bez else.
* Katrai funkcijas jādara viens darbs, bet labi.
* Katrai funkcijai kaut kas ir “jānoslēpj”.
* Nelabojiet sliktu programmu – pārrakstiet to.
* Lielu programmu rakstiet un pārbaudiet pa mazām daļām.

1. Ievade un izvade

* **Pārbaudiet ieejas datus.**
* Paziņojiet par ievades kļūdām.

1. Dažādi ieteikumi

* Inicializējiet mainīgos pirms to izmantošanas.
* Nelietojiet vairākas izejas no cikla.
* Pārbaudiet programmu uz robežvērtībām.
* Nepārbaudiet peldošus skaitļus uz vienādību, jo tie aptuveno darbību dēļ var tieši nesakrist.

**Specifikācija** ir detalizēts ierīces, procesa vai izstrādājuma nepieciešamo raksturojumu precīzs apraksts.

Piemērs funkcijas specifikācijai:

int cikdazad(char\* rin);

Funkcija cikdazad(rin) -

atgriež kā rezultātu dažādo simbolu skaitu

zema līmeņa simbolu virknē rin.

# Datu izvade C++ konsoles režīmā

Tiek izmantota bibliotēka <iostream>. Izvade notiek, izmantojot objektu cout un izvades operatoru <<. Pēc noklusējuma izvade notiek vienā rindiņā, lai pārietu jaunā rindiņā, vajag izvadīt vērtību endl, vai izvadīt \n simbolu. Lai izvadītu vairākas vērtības, tās vajag atdalīt ar izvades operatoru. Izvadi ir iespējams formatēt ar <iomanip> bibliotēku, kur ir iespējams iestatīt skaitļu ar komatu izvadīto komatu skaitu, cik vietas aizņems kāda vērtība u.c. Formatēšana var tikt veikta ar divu mehānismu palīdzību: izmantojot manipulatorus (izdrukas priekšraksta ietvaros) un izmantojot objekta cout funkcijas (pirms izdrukas priekšraksta).

cout << setprecision(5) << 271.84753; // 271.85

cout.precision(4); cout << 271.84753; // 271.8

cout << setw(3) << "C"; // C

cout.width(12); cout << right << "D"; // D

cout << scientific << 271.84753; // 2.718e+002

# Datu ievade C++ konsoles režīmā

Tiek izmantota bibliotēka <iostream>. Ievade notiek, izmantojot objektu cin un ievades operatoru >>. Ievade tiek uzkrāta buferī līdz tiek nospiests Enter, kad no ievades mēģina atbilstoši mainīga tipam mainīgajam piešķirt ievades vērtību. Var veikt vērtību piešķiršanu vairākiem mainīgajiem, izmantojot cin un starp katru mainīgo liekot ievades operatoru. Ievadē dažādās vērtības ir jāatdala ar atdalītājsimbolu (atstarpe, pārnese jaunā rindā u.c.). Ievades process notiek 2 posmos:

* informācijas nolasīšana no klaviatūras un uzkrāšana buferī līdz pat ENTER nospiešanai,
* bufera satura pārstaigāšana, mēģinot izvilkt noteikta tipa vērtību vai vērtības, lai ierakstītu mainīgajā vai mainīgajos.

Lai ievadītu simbolu virkni (kas var iekļaut pašu atdalītājsimbolu), jāizmanto objekta cin metode getline, norādot mainīgo un maksimālo garumu. cin.getline(var,len) automātiski galā ievietos pārnesi jaunā rindā. Funkciju getline(cin,var) izmanto augsta līmeņa simbolu virknēm (string), kurām kompilācijas laikā nav zināms garums. Pirms to lietošanas lietot cin.ignore(), lai ignorētu buferī uzkrāto ievadi.

cin >> i >> k; // 1 [space] 2 [enter]

# Datu ievade un izvade Python

Datu izvade notiek ar funkciju print(). Vairākas vērtības var izdrukāt, tās uzrādot kā argumentus, to atdalītājsimbolu var nomainīt ar argumentu sep=”sym”. Tās beigās tiek automātiski izvadīts jaunas rindas simbols, ko var nomainīt ar end=”sym”. Datus var noformatēt izmantojot metodi format iekš funkcijas print. Var arī kādu vērtību izvadīt vairākas reizes to print funkcijā sareizinot. Formatēšanai izmanto papildus metodi format, kas tiek piemērota formatējumam, kur figūriekavu pāri nozīmē vietas, kur ievietot datus, bet paši dati padoti kā parametri. Izmantojot papildus modifikatoru f, var noteikt ciparu skaitu aiz komata. Teksta formatēšanā var izmantot, ka operators \* nozīmē teksta atkārtošanu konkatenējot.

d=12.3456789

print("{0:.3f}".format(d)) // 12.346

print("{0:.3e}".format(d,d)) // 1.235e+01

print("#"\*10) // ##########

Ievade notiek ar funkciju input(). Var padot arī parametrā uzaicinošo tekstu (*prompt*). Tiek nolasīta tikai 1 vērtība, no kuras var iegūt vairākas to uzreiz vai pēc tam apstrādājot, piemēram ar metodi split(). Ievade valodā Python notiek tikai teksta formā (t.i., lai iegūtu, piemēram, skaitli, ir jāveic tieša pārveidošana no teksta uz skaitli), piemēram, num=int(input()).

# Programmēšanas valodas leksiskā līmeņa elementi – identifikatori (identifiers), atslēgas vārdi (keywords), literāļi (literals), operatoru marķieri (operator tokens), dienesta simboli (punctuators)

* **Identifikatori** ir mainīgā vai cita programmēšanas elementa vārds. Tas var būt funkcijas, klases, objekta vārds. Rezervētajiem vārdiem valodā C++ ir noteikta nozīme, kuru nedrīkst mainīt, tomēr saviem mainīgajiem un citiem programmas elementiem programmētājs var brīvi izvēlēties identifikatorus, tomēr tie nedrīkst konfliktēt ar rezervētajiem vārdiem un citiem identifikatoriem. Identifikators var sākties ar burtu vai pasvītrojuma zīmi, bet pārējie simboli var būt arī cipari. C++ identifikatoru pieraksts ir reģistrjūtīgs.
* **Atslēgas vārdi** ir programmēšanas valodā iekšēji definēti vārdi. To nozīme ir nemaināma. Atslēgas vārdus apraksta identifikatori. Atslēgas vārdi apraksta iebūvētos datu tipus, dažādas konstrukcijas un citus valodas elementus. Piemēram, if, return, continue, break, int, using namespace utml.
* **Literāļi** ir dažādu tipu vērtības, kas tiešā veidā parādās programmas tekstā. Literāļi var būt veseli skaitļi, skaitļi ar peldošo komatu, simboli, simbolu virknes un loģiskās vērtības.
* **Operatora marķieri** ir simboli, kas veic kādu darbību, kad tie ir pievienoti pie kāda mainīgā. Mainīgos, ar kuriem šīs darbības tiek veiktas, sauc par operandiem.
* **Dienesta simboli** ir simboli, kas kaut ko simbolizē, piemēram, sākums un beigas funkcijas ķermenim – figūriekavas { }, funkcijas parametru sākums un beigas – iekavas ( ), komandas beigas – semikols ; utml.

# Mainīgie (variables) un datu tipi (data types) valodā C++

**Mainīgais ir vieta datora atmiņā, kur tiek glabāti noteikta tipa dati. Inicializācija ir mainīgā pirmreizējā aizpildīšana ar vērtību.**

Mainīgo programmā identificē noteikts vārds – identifikators. Pirms sākt strādāt ar mainīgo, tas ir jāinicializē. Nevajadzētu izmantot automātisku mainīgo inicializāciju, ko veic dažas izstrādes vides. Mainīgajam var piešķirt vērtību arī pēc inicializācijas.

Īpašs mainīgo veids (no izmantošanas viedokļa) ir **norādes mainīgie**, kuros glabātā vērtība ir kāda cita atmiņas apgabala adrese. Šajā gadījumā var teikt, ka mainīgais reprezentē atmiņas apgabala sākumu, kurā glabāt datus.

No atmiņas izmantošanas viedokļa mainīgos iedala: **statiskie** un **dinamiskie**.

No redzamības viedokļa dažādos programmas moduļos mainīgos iedala: **globālie** un **lokālie**.

**Datu tips (no datu apstrādes viedokļa) ir darbību kopums, ko var veikt ar noteiktu vērtību.** Katru datu tipu raksturo atmiņas daudzums, ko aizņem viena šī tipa vērtība, vērtību diapazons, kas atbilst šim datu tipam, un darbības, ko ar to var veikt.

Tipiskākais piemērs valodā C++ tam ir dalīšana. Ir parastā dalīšana (piemēram, 16.0/5=3.2) un veselo skaitļu dalīšana (16/5=3, atlikumā 1), kas ir pilnīgi cita operācija, tomēr abas šīs dalīšanas operācijas tiek apzīmētas ar ‘/’, un to, kura no tām izpildīsies, nosaka tikai un vienīgi operandu tipi, uz ko attiecas dalīšana.

Pastāv datu pamattipi: skaitliskie datu tipi: int (4 baiti), float (4 baiti) double (8 baiti), loģiskie datu tipi: bool (1 baits) un simboliskie datu tipi: char (1 baits). No datu pamattipiem var būvēt saliktus datu tipus – masīvus, struktūras, klases.

Tipu pārveidošana ir process, lai pārveidotu viena datu tipa vērtību uz citu, pielāgojot šo vērtību jaunajam datu tipam.

# Mainīgie (variables) un datu tipi (data types) valodā Python

**Mainīgais (variable) ir mehānisms, kas programmā reprezentē noteiktu vērtību. Inicializācija ir mainīgā pirmreizējā aizpildīšana ar vērtību.**

Lai ar mainīgo veiktu darbības, to obligāti vajag inicializēt. Nepastāv nodalīta mainīgā deklarācijas komanda, jo tas notiek automātiski, piešķirot mainīgajam vērtību.

Python valodā tiek lietota mainīgo saistīšana (*binding*). **Piešķiršana** ir mainīgā uzstādīšana norādīšanai uz noteiktu vērtību. Valodā Python mainīgais savā ziņā ir tikai nosaukums, kas norāda uz kādu vērtību. Mainīgais programmas izpildes laikā var atsevišķos laika posmos reprezentēt dažādu tipu vērtības.

No redzamības viedokļa dažādos programmas moduļos mainīgos iedala globālajos un lokālajos.

**Datu tips (no datu apstrādes viedokļa) ir darbību kopums, ko var veikt ar noteiktu vērtību.** Katru datu tipu raksturo atmiņas daudzums, ko aizņem viena šī tipa vērtība, vērtību diapazons, kas atbilst šim datu tipam, un darbības, ko ar to var veikt.

Tādējādi it kā viena un tā pati darbība ar dažādu datu tipu vērtībām var tikt veikta dažādi. Piemēram operācija ‘+’ skaitliskajiem datu tipiem nozīmē saskaitīšanu, bet simbolu virknēm - konkatenāciju.

Python valodā mainīgajam nav datu tipa, tas ir tikai mainīgā vērtībai. int ir nenoteikta izmēra, float ir 8 baiti, str ir atkarīgs no simbolu skaita un izmēra, bool tāpat kā int.

# Mainīgo deklarēšana, inicializācija un piešķiršana

**Mainīgā deklarēšana** paziņo, ka noteikts programmas elements tiks izmantots programmā. Deklarācija ietver mainīgā vārdu, tipu, un neobligāti – inicializāciju. **Inicializācija** ir mainīga pirmreizējā aizpildīšana ar vērtību. To var apvienot ar deklarēšanu. **Piešķiršana** ir vēlāka vērtības ierakstīšana mainīgajā. To norāda piešķiršanas operators – vienādības zīme. Vairāku piešķiršanu gadījumā, piešķiršana notiek no labās uz kreiso pusi. Arī piešķiršanas komandai ir vērtība. Saīsinātā piešķiršanas komanda.

int a; // deklarācijas

a = 15; // inicializācija

int b, l=17; // mainīgo deklarācija, mainīgā inicializācija

cin >> b; // inicializācijas komanda

a = b++; // salikta piešķiršanas komanda

a = b = 7; // salikta piešķiršanas komanda

char d = ++a; // deklarācija & inicializācija

a = 15; // inicializācija bez deklarācija Python valodā

a = {1, 5}; // piešķiršana ar mainīgo saistīšanu

b = input(); // inicializācija bez deklarācija Python valodā

# Veselo skaitļu datu tips (int) un loģiskais datu tips (bool). Darbības un vērtību pieraksts

* Datu tips int ir vesels skaitlis jeb skaitlis bez komata, tas aizņem 4 baitus lielu atmiņu, tā vērtību diapazons ir no –231 līdz 231 – 1. Ar to var veikt aritmētiskas darbības.
* Datu tips bool ir patiesumvērtība, tas aizņem 1 baitu lielu atmiņu, tā vērtība ir vai nu true (skaitlis, kas nav 0), vai nu false (skaitlis 0). Ar to var veikt loģisko izteiksmju darbības.

# Peldošā komata skaitļu datu tips (float, double). Darbības un vērtību pieraksts

* float ir skaitlis ar peldošo komatu, kas aizņem 4 baitus lielu atmiņu, tā vērtību diapazons ir no aptuveni no 10-38 līdz 1038, ar to var veikt aritmētiskas darbības.
* double arī ir skaitlis ar peldošo komatu, kas aizņem 8 baitus lielu atmiņu, tā vērtību diapazons ir no aptuveni 10-308 līdz 10308, un ar to arī var veikt aritmētiskas darbības.

Skaitlis ar peldošo komatu sastāv no veselās daļas un mantisas (daļas pēc decimālās zīmes), kam var sekot desmitnieku pakāpe, ko ievada simbols e vai E: -1.2e-3 nozīmē jeb . Vienu no divām – veselo daļu vai mantisu var izlaist, tādējādi 0.3 var pierakstīt kā .3 , bet 4.0 kā 4.

Piemēram, 1.0, -0.0, -.4, 3., 1.23e4, 2.e-3, .5e+2, 6E1

# Veselo un peldošā komata skaitļu apstrādes īpatnības

Apstrādājot mainīgos ar šiem datu tipiem, rezultāts arī būs šāds mainīgais, pat, ja tiek dalīts ar int tipa mainīgo. Šos mainīgos dators uzglabā normālformā, kur ir bāze un pakāpe. Tā kā ir ierobežots bitu skaits bāzes vērtībai, vērtību izkliede nav vienmērīga, t.i. nevar attēlot visus reālos skaitļus intervālā no aptuveni 10-308 līdz 10308, bet gan to tuvinājumus. Turklāt vērtību izkliede ir nevienmērīga. Tādējādi var rasties kļūdas, kad kādu no pēdējiem cipariem noapaļo. double ir lielāks bitu skaits, tādēļ tam ir lielāka precizitāte.

# Mainīgo redzamība un eksistences periods. Lokālie un globālie mainīgie

Programmas daļu, kurā ir redzams noteikts mainīgais, sauc par šī mainīgā redzamības apgabalu (*scope*). Bloks ir programmas daļa, kas ietverta starp atverošo un aizverošo figūriekavu vienas funkcijas ietvaros.

**Lokālie mainīgie** **ir tādi mainīgie, kas darbojas un ir redzami tikai noteikta bloka (piemēram, funkcijas) iekšienē.** Šādiem mainīgajiem dzīves ilgums ir no tā deklarēšanas līdz bloka izpildes beigām.

{

int a; // sākot ar šo vietu, mainīgo a var izmantot

...

} // te mainīgais a vairs nav redzams

**Globālos mainīgos** var redzēt visa programma, to dzīves ilgums ir līdz programmas beigām. Globālos mainīgos parasti izmanto tikai specifiskos gadījumos, jo to izmantošana ir pretrunā ar procedurālo abstrakciju un parasti uzskatāma par sliktu stilu. Globālais mainīgais tiek deklarēts ārpus jebkura bloka un tādēļ ir redzams jebkura bloka iekšienē.

# Skaitliskas izteiksmes (numeric expressions)

***Izteiksme ir konstrukcija, kas sastāv no operatoriem un operandiem un nosaka skaitļošanas procesu.***

Aritmētiska izteiksme (*numeric expression*) ir konstrukcija, kas sastāv no skaitliskām operācijām un skaitliskiem operandiem, un kura pēc šo operāciju izpildes izrēķina skaitlisku vērtību. Skaitliskie operatori ir, piemēram, +, -, /, \*, %, operandi ir skaitliski mainīgie vai literāļi, kam tiek veiktas šīs darbības.

Atšķirībā no matemātikas, pieraksta lineāri, izpildes secību ietekmējot ar papildus iekavām.

(sin(x) + 1) / y + y / (x + 1.5)

Svarīgi atcerēties, ka divas dažādas dalīšanas – veselo skaitļu dalīšana un parastā dalīšana tiek apzīmētas ar vienu un to pašu operāciju /. To, kura no darbībām tika izpildīta, nosaka konteksts – ja abi operandi ir veseli skaitļi, tiek izpildīta veselo skaitļu dalīšana, bet ja kaut viens tāds nav – tad parastā dalīšana. Atlikuma operācija izpildāma tikai veseliem skaitļiem. Pārējās darbības (+ - \*) attiecas un vienādi darbojas gan ar veseliem skaitļiem, gan skaitļiem ar peldošo komatu, atšķirības ir tikai vērtības tipā, kas tiek atgriezts pēc dotās operācijas izpildes – ja abi operandi ir veseli skaitļi, atgrieztās vērtības tips arī ir vesels skaitlis, bet, ja kaut viens ir skaitlis ar peldošo komatu, tad arī atgriežamā vērtība ir skaitlis ar peldošo komatu.

# Loģiskas izteiksmes (logical expressions)

***Izteiksme ir konstrukcija, kas sastāv no operatoriem un operandiem un nosaka skaitļošanas procesu.***

**Loģiska izteiksme ir konstrukcija, kas sastāv no loģiskām operācijām un loģiskiem operandiem, un kura pēc šo operāciju izpildes izrēķina loģisku vērtību – patiess vai aplams.**

Loģiska izteiksme ir zarošanās priekšraksta if-then-else, kā arī visu cikla priekšrakstu obligāta sastāvdaļa.

**Salīdzināšanas operatori** salīdzina divas vērtības (<, >, ==, <=, >=, !=).

Dažādas loģiskās vērtības, piemēram, salīdzināšanas operācijas, var apvienot, veidojot sarežģītākas loģiskās izteiksmes. Šim nolūkam der loģiskie operatori ! && ||. Loģisko operatoru operandi var būt jebkādas loģiskas vērtības, t.sk., salīdzināšanas operācijas, funkcijas, t.i., citas loģiskas izteiksmes. Loģiskos operatorus var kombinēt. Pastāv loģisko operatoru prioritātes rinda, un, lai nomainītu izpildes secību, jālieto papildus iekavas. Patiesumvērtību tabula AND, OR, NOT.

Valodā C++ pēc būtības nav atsevišķa loģiska tipa un loģisku vērtību. Tips bool un vērtības true, false ieviestas tikai jaunākajā C++ standartā. Pēc būtības tips bool ir tas pats int viena baita garumā, true ir 1, bet false – 0, un ne tikai pēc būtības atbilst, bet ir līdzvērtīgi lietojami. Loģiskās operācijas darbojas vēl plašāk – 0 apzīmē false, bet jebkura cita vesela skaitļa vērtība (piemēram, 1, -2, 3) – true. Ņemot vērā to, ka loģiskas vērtības ir skaitliskas vērtības, tad var uzskatīt, ka jebkura loģiska izteiksme ir arī aritmētiska izteiksme, bet jebkura aritmētiska izteiksme – arī loģiska, turklāt drīkst brīvi būvēt jauktu aritmētiski-loģisku izteiksmi.

((m<=3 || m>10) && d==30)

# Skaitlisku vērtību apstrāde C++ un Python

Skaitlisku vērtību apstrādei izmanto aritmētiskas izteiksmes vai funkcijas.

Aritmētiska izteiksme ir konstrukcija, kas sastāv no skaitliskām operācijām un skaitliskiem operandiem, un kura pēc šo operāciju izpildes izrēķina skaitlisku vērtību. Operatori ir, piemēram, +, -, /, \*, %, operandi ir skaitliski mainīgie vai literāļi, kam tiek veiktas šīs darbības.

Ja vēlas izmantot kāpināšanu, kvadrātsakni, noapaļošanu vai trigonometriskas funkcijas, nepieciešams patstāvīgi uzbūvēt atbilstošu funkciju vai izmantot kādu bibliotēku, piemēram, #include <math.h> ar jau gatavām funkcijām.

pow(4, 0.5) // 2 kāpināšana 4 pakāpē 0.5

ceil(-4.5) // -4 noapaļošana uz augšu

floor(-4.5) // -5 noapaļošana uz apakšu

round(-4.5) // -5 noapaļošana matemātiski

fabs(-4.5) // 4.5 absolūtā vērtība (bez zīmes)

Ja darbības tiek veiktas ar viena tipa mainīgajiem, tad rezultāts būs tā tipa, bet, ja tiek veikts ar atšķirīgiem tipiem, tad rezultāts pieņems vienu no tipiem. Tam ir noteikta kārtība, piemēram, float pārveido par double, bet int – par float.

Valodā Python ar import math. Kāpināšana, izmantojot operatoru \*\*. Veselo skaitļu dalīšana tikai ar //.

math.ceil (num);

math.floor (num);

round (num);

math.sin (arg);

math.log (arg);

# Zarošanās priekšraksts if-else. Vairāku līmeņu zarošanās

**Zarošanās ir viena no trim galvenajām vadības konstrukcijām. Viens no veidiem, kā valodā C++ realizē zarošanos, ir zarošanās priekšraksts if-else*.***

* if-else izmanto, ja programmas darbības gaitā rodas nepieciešamība atkarībā no kāda nosacījuma izpildīt vai neizpildīt kādu programmas daļu.
* if-else priekšraksts sastāv no trim blokiem, no kuriem trešais ir neobligāts:
  + izvēles nosacījums, kas ir loģiska izteiksme, visbiežāk salīdzināšanas operācija,
  + darbība, kas tiek veikta, ja nosacījums izpildās (then bloks),
  + (neobligāti) darbība, kas tiek veikta, ja nosacījums neizpildās (else bloks).

Formāli valodā C++ operatoram if-else nav speciāli izdalīta daļa else if. Ja programmā parādās else if, tad attiecīgais if tiek uzskatīts par augšējā if-else priekšraksta else zara sastāvdaļu. Tomēr praksē ir daudz vieglāk to uztvert kā kaskādes veida konstrukciju, nevis kā vairāku līmeņu konstrukciju.

if (a > 0) cout << "positive" << endl;

else cout << "not positive" << endl;

if (a > 0) cout << "positive" << endl;

else if (a == 0) cout << "zero" << endl;

else cout << "negative" << endl;

Komplektā ar to, ka if-then-else priekšraksts var parādīties 2 veidos – ar un bez else zara – tas var it kā novest pie efekta, ka nav skaidrs, kuram if-then-else priekšrakstam īsti pieder else zars. Tāpēc, lai novērstu divdomību, ir noteikts, ka else attiecas uz tuvāko if. Sekojošajā piemērā else attiecas uz if (a > 0). Tādējādi, dotais programmas kods vispār kaut ko izdrukā tikai tad, ja a ir pāra skaitlis.

if (a%2 == 0)

if (a > 0) cout << "Positive even" << endl;

else cout << "Negative even" << endl;

# Zarošanās priekšraksts switch un nosacījuma funktors

Zarošanās (*branching, selection*) ir viena no trim galvenajām vadības konstrukcijām (blakus secībai (*sequence*) un ciklam (*looping*). Zarošanos valodā C++ realizē zarošanās priekšraksti switch-case.

Ja izvēle notiek starp diskrētām vērtībām, kas nav īpaši lielā skaitā, tad ērtāk un uzskatāmāk būtu lietot priekšrakstu switch-case. Priekšraksta switch-case darbības princips ir lielā mērā līdzīgs if-else darbībai ar vienu būtisku izņēmumu – pēc atbilstošās darbības izpildes pie dotās izvēles programmas vadība automātiski netiek nodota uz priekšraksta beigām, bet gan nonāk nākošās izvēles apstrādes blokā, ja vien netiek lietots operators break. Tas nodrošina iespēju vairākām izvēles iespējām rakstīt vienu vienīgu apstrādes kodu.

switch (a) {

case 6:

cout << "Saturday" << endl;

break;

case 7:

cout << "Sunday" << endl;

break;

default:

cout << "Working day" << endl;

};

Nosacījuma funktors ?: darbojas pēc līdzīga principa kā if-else priekšraksts ar atšķirību, ka veicamo darbību (priekšrakstu) vietā ir izteiksmes, kas atgriež vērtību, tādējādi arī pats funktors atgriež noteiktu vērtību – to, kuru izrēķina attiecīgā izteiksme.

cout << (a>0 ? "positive" : "not positive") << endl;

# Cikla priekšraksti – for, while, do-while

**Cikls ir vadības konstrukcija, kas atkārto noteiktas darbības līdz brīdim, kad pārstāj izpildīties noteikts nosacījums.** Jebkuram ciklam ir nosacījums un ķermenis. Cikla ķermenis ir programmas bloks cikla konstrukcijas ietvaros, kas nosaka darbību, kas ciklā tiks atkārtota. Viena cikla ķermeņa izpildīšanu vienu reizi sauc par iterāciju.

1. Ciklā ar skaitītāju pirms cikla izpildes jau ir zināms atkārtošanas reižu skaits, un tas uzskatāms par cikla ar nosacījumu speciālgadījumu. Operators for ir ērtāks cikla ar skaitītāju konstruēšanai.

* Cikls for ar skaitītāju sākas ar sākuma stāvokļa uzstādīšanu, kurā parasti tiek inicializēti (vai pat deklarēti un inicializēti) mainīgie. Pēc tam notiek cikla nosacījuma pārbaude. Ja pārbaude ir veiksmīga (true), tiek izpildīta kārtējā iterācija, kas sastāv no cikla ķermeņa izpildes, kam seko cikla soļa izpilde. Pēc katras iterācijas tiek pārbaudīts cikla nosacījums. for cikla galvā visas 3 komponentes ir neobligātas – ja nav uzrādīta otrā komponente, tad vienīgais veids, kā iziet no cikla, ir operators break.
* Sākot ar C++11 versiju, ir pieejams vēl viens cikla for veids – datu virknes pārstaigāšana, kur katru cikla iterāciju reprezentē nevis skaitītāja vērtība, bet gan elements datu virknē.

for (int m: {1,2,12}) cout << m << " " << m\*m << endl;

1. Cikls while ir funkcionāli identisks ciklam for ar divām sintaktiskām atšķirībām – sākuma stāvokļa uzstādīšana iznesta pirms cikla, bet cikla solis var tikt novietots cikla ķermeņa beigās.
2. Cikls do-while atšķiras no while ar to, ka ieeja ciklā notiek uzreiz cikla ķermenī (nevis caur cikla nosacījumu), tādējādi nodrošinot, ka cikla ķermenis tiks izpildīts vismaz vienu reizi. Atšķirībā no cikla while, cikla ķermenis ne tikai atrodas pirms nosacījuma, bet tas nedrīkst būt tukšs.

# Izeja no cikla, apejot cikla nosacījumu – continue, break, return

* continue nodrošina kārtējās cikla iterācijas pārtraukumu un pāreju uz nākamo iterāciju, izlaižot atlikušās cikla ķermeņa daļas izpildi.
* break nodrošina kārtējās cikla iterācijas pārtraukumu un izeju no cikla.
* return atgriež funkcijas vērtību, tādēļ gan pārējais kods ciklā, gan arī ārpus tā vairāk netiks pildīts.

Teorētiski tā ir neliela atkāpšanās no strukturētas programmēšanas pamatprincipiem.

# C++ programmas vispārējā struktūra (funkcijas, bloki, priekšraksti)

Konstrukciju līmenī varētu teikt, ka C++ programmas sastāvā ietilpst:

**Priekšraksts (statement)** **ir pabeigta valodas konstrukcija, kas veic noteiktu darbību.** Varētu teikt, ka no konstrukciju viedokļa programma sastāv pamatā no priekšrakstiem. Saliktu priekšrakstu piemēri ir izvēles un cikla priekšraksti, bet vienkāršu priekšrakstu piemēri – piešķiršanas, funkcijas izsaukuma, atdalīšanas priekšraksti.

Deklarators (declarator) ir konstrukcija, kas paziņo, ka programmā tiks izmantots noteikts elements (visbiežāk mainīgais) vai elementi, iespējami uzstādot šim elementam sākotnējās vai noklusētās vērtības.

**Bloks (block) ir priekšrakstu vai deklaratoru virkne, kas ietverta figūriekavās.**

Par moduli (module) šajā mācību materiālā tiek saukts patstāvīgs programmas bloks. Moduli ievada moduļa galva, kas lielā mērā to identificē. Tipiskākie moduļu piemēri valodā C++ ir funkcijas, klases un datu struktūras. Procedurālā abstrakcija ir programmas strukturēšanas veids, kad programma tiek strukturēta moduļos (procedūrās) pēc “melnās kastes” principa, t.i., nodalot redzamo vispārīgo (deklaratīvo) daļu no paslēptās detalizētās (realizācijas) daļas.

**Funkcija ir patstāvīgs programmas bloks, kas veic noteiktas darbības, un atgriež noteikta tipa vērtību. Funkciju raksturo vārds, atgriežamais tips un parametri.**

Vienkāršākajā gadījumā C++ programma ir funkciju kopums, kurā tieši viena saucas main. Tieši no funkcijas main sākās C++ programmas izpilde. (*Izņēmums ir vienīgi Windows specifiskas programmas, kur programmu ievadošā funkcija tiek saukta WinMain*) Funkcijas satur dažādus priekšrakstus, piemēram, if-else, kas atkarībā no loģiskas izteiksmes vērtības izpilda vai nu if bloku, vai else bloku.

# C++ programmas failu struktūra un kompilēšana

Jebkura C++ programma sastāv no vismaz viena programmas faila (.cpp), kurā ir main funkcija. Elementu (mainīgo, funkciju, klašu) deklarācijas liek atsevišķos hedera failos (.h), un tie tiek realizēti vēl citos programmas failos. Galvenajā C++ failā iekļauj tikai hedera failus ar direktīvas #include palīdzību. To, ka dažādi C++ faili veido vienotu programmu, realizē projektu mehānisms attiecīgajā izstrādes vidē.

**1. fāzē** katram C++ failam tiek izveidots objektu fails, veidojot objektu kodu.

**2. fāzē** notiek linkošana, kur no objekta koda tiek izveidots izpildes kods, parasti izpildāms (.exe) fails.

**add.h**

int add (int, int); // funkcijas deklarācija

**add.cpp**

#include "add.h"

int add (int a, int b) // funkcijas realizācija

{

return a + b;

};

**addmain.cpp**

#include "add.h"

#include <iostream>

using namespace std;

int main ()

{

int x=5, y=7, z;

z = add (x, y);

cout << z << endl;

return 0;

}

# Python programmas vispārējā struktūra

**Valodas Python programma ir konstrukciju kopums, kas apraksta noteiktu algoritmu.** Python programma no valodas konstrukciju viedokļa ir dažādu komandu un nosaukto bloku (funkciju un klašu) kopums. Programmas darbināšana sākas no faila sākuma, ejot pa (pirmā hierarhijas līmeņa, t.i., bez atkāpes) elementiem.

1. sastopot komandu, tā tiek izpildīta; ja šī komanda ir moduļa ielādes komanda import, tad tiek izsaukta attiecīgā moduļa (t.i., faila) darbināšana pēc šī paša principa – tātad, ielādējot moduli, tur esošās “parastās” komandas tiek izpildītas, nevis ielādētas,
2. sastopot nosauktu bloku (funkciju vai klasi), tas tiek ielādēts.

Attēls, kurā ir teksts, diagramma, ekrānuzņēmums, rinda

Apraksts ģenerēts automātiski

# Funkcija C++ – deklarācija, definīcija, izsaukums

**Funkcija ir patstāvīgs programmas bloks, kas veic noteiktas darbības, un atgriež noteikta tipa vērtību.**

Parametri ir mehānisms, ar kuru vērtības tiek nodotas funkcijai noteiktu darbību veikšanai.

Argumenti ir vērtības, kas, funkciju izsaucot, tai tiek padotas caur parametriem.

**Funkcijas deklarācija ir tāda informācija par funkciju, kas ietver funkcijas vārdu, parametru skaitu un tipus, kā arī atgriežamo tipu, kas ir paziņojums par to, ka funkcija programmā tiks izmantota**. Funkcijas deklarācijai, ja tā vispār tiek lietota, jāatrodas gan pirms realizācijas, gan pirms izsaukuma. Funkcijas deklarācija *var* saturēt parametru vārdus un noklusētās vērtības.

**Funkcijas definīcija ir pilns funkcijas apraksts, kas sastāv no funkcijas galvas, kas ir līdzīga prototipam, un funkcijas ķermeņa.** Funkcijas galvā atšķirībā no prototipa ir obligāti jāuzrāda parametru nosaukumi. Funkcijas ķermenim, ja tā atgriežamais tips nav void, obligāti jāsatur vismaz viens atgriešanas priekšraksts.

**Funkcijas izsaukums (*function call*) ir izpildāmā uzdevuma nodošana funkcijai, vajadzības gadījumā (un ja atgriežamais tips nav void) iegūstot funkcijas izrēķināto vērtību.**

* Funkcijas deklarācijai ir jābūt pirms realizācijas un izsaukuma;
* Realizācijas un izsaukuma savstarpējai secībai nav nozīmes;
* Funkcijas deklarācija var tikt izlaists, tādā gadījumā tā lomu pilda realizācija.

int add (int, int); // funkcijas deklarācija

int add (int a, int b) { // funkcijas realizācija

return a + b;

}

# Funkcijas parametri (parametri-vērtības, parametri-references, noklusētie, konstantie) un atgriežamais tips C++

1. **Parametrs-vērtība** ir tāds parametrs, caur kuru padotais arguments tiek dublēts funkcijas vajadzībām. Tā kā funkcija tādā gadījumā strādā ar padotās informācijas kopiju, tad izsaucošajā funkcijā attiecīgā informācija netiek izmainīta.
2. **Parametrs-reference** ir tāds parametrs, caur kuru padotais arguments netiek dublēts, bet nonāk koplietošanas režīmā starp izsaucošo un izsaucamo funkciju. Lai parametru apzīmētu kā references parametru, to deklarējot (un definējot) aiz parametra tipa jāliek references simbols &.

**Noklusētie parametri** ir parametri, kurus funkcijas izsaukumā var izlaist, un kas izlaišanas gadījumā pieņem iepriekš noteiktas vērtības. Noklusētie parametri nedrīkst atrasties pa kreisi no nestandarta parametriem. Noklusētie parametri vai nu funkcijas deklarācijā vai definīcijas galvā ietver noklusētās vērtības uzstādīšanu, tomēr parasti to dara tieši deklarācijā – abās vietās to darīt nedrīkst.

**Konstantie parametri** ir tādi parametri, caur kuriem padotos argumentus nedrīkst izmainīt funkcijas iekšienē. Parametra konstantums tiek norādīts ar modifikatora const palīdzību pirms parametra tipa.

void drukat (const string &a, int b=10);

# Funkcija Python

Procedurālā abstrakcija ir programmas strukturēšanas veids, kad programma tiek strukturēta moduļos (procedūrās) pēc “melnās kastes” (black box) principa, t.i., nodalot redzamo deklaratīvo daļu no paslēptās definīcijas daļas.

**Funkcija ir patstāvīgs nosaukts programmas bloks, kuru iespējams izsaukt no citas koda daļas un kas ar padotajiem datiem veic noteiktas darbības un iespējami atgriež vērtību. Ārēji funkciju raksturo vārds un parametri.**

def add(a,b):

return a+b

Funkcijas ķermenī var tikt veidoti mainīgie, un tie būs t.s. lokālie mainīgie – izmantojami tikai funkcijas iekšienē. No funkcijas ķermeņa skatupunkta arī funkcijas parametri ir lokālie mainīgie. Atšķirībā, piemēram, no C++, valodā Python mainīgo lokālums ir visas funkcijas līmenī (nevis mazāka struktūras bloka līmenī).

Funkcijas izsaukumā tiek izmantoti visi vai daļa no parametriem, kas noteikti funkcijas definīcijā (daļa parametru ir uzstādāmi kā noklusētie). Funkcija var būt bez parametriem un arī neko neatgriezt. Python funkcija vienmēr kaut ko atgriež – ja nekas nav noteikts, tad speciālo vērtību None.

* ja parametru funkcijā izmanto lasīšanas režīmā, vai caur to tiek modificēta maināma (*mutable*) vērtība, piemēram, list, tad tas strādā references režīmā.
* ja parametra mainīgajam tiek veikta piešķiršana, tad tiek veidota jauna vērtība, un parametrs “pārtop” par vērtības parametru.

Funkciju izvietošanai programmā ir šādi nosacījumi:

* funkcijas definīcijai jābūt pirms tās izsaukuma
* drīkst ievietot funkciju (funkcijas definīciju) citā funkcijā
* modulī drīkst būt tikai viena funkcija ar šādu nosaukumu (valodā Python
* nav funkciju pārslogošanas)

# Funkciju pārslogošana un noklusētie parametri

**Pārslogošana (*overloading*) ir vienāda vārda izmantošana dažādu funkciju nosaukšanai.** Pārslogošana ir polimorfisma apakšgadījums. Funkciju pārslogošana valodā C++ iespējama tāpēc, ka funkcijas tiek identificētas nevis tikai ar vārdu, bet papildus arī ar to parametru tipu virkni, signatūru.

**Funkcijas signatūra valodā C++ ir funkcijas apraksts, kas ietver funkcijas vārdu un parametru tipu virkni, ieskaitot arī parametru modifikatorus const. Funkcijas signatūra identificē funkciju programmā.**

float add (float a, float b); // funkcijas deklarācija

double add (double a, double b); // funkcijas deklarācija

int x=3, y=4;

cout << add (x, y); // kļūda

cout << add ((double)x, (double)y); // nav kļūdas

**Kompilatora kļūda: Neviennozīmīga pārslogotas funkcijas izsaukšana (divdomība)**

**Noklusētie parametri** ir parametri, kurus funkcijas izsaukumā var izlaist, un kas izlaišanas gadījumā pieņem iepriekš noteiktas vērtības. Noklusētie parametri nedrīkst atrasties pa kreisi no nestandarta parametriem. Noklusētie parametri vai nu funkcijas deklarācijā vai definīcijas galvā ietver noklusētās vērtības uzstādīšanu, tomēr parasti to dara tieši deklarācijā – abās vietās to darīt nedrīkst.

# Funkciju šablons C++ un parametru dinamisms Python

**Funkciju šablons ir shēma funkciju veidošanai.**

* funkciju šablona izveidošana ir līdzīga parastas funkcijas izveidošanai ar nekonkretizētiem datu tipiem, kam pievienota šablona galva,
* funkciju šablona izmantošana pilnīgi neatšķiras no parastas funkcijas izmantošanas.

Jāatceras, ka funkciju šablons nav universāla funkcija (kaut arī no izmantošanas viedokļa tā varētu likties), bet shēma funkciju izveidošanā, t.i. uz tā bāzes tiek izveidots tik daudz “īstu” funkciju, cik nepieciešams.

* funkciju šablona (arī klašu šablona) izmantošana nodrošina efektīva izpildāmā koda ģenerēšanu, jo katram gadījumam tiek izveidota sava funkcija (vai klase),
* tajā pašā laikā šabloni šī paša iemesla dēļ nenodrošina kompakta izpildāmā koda ģenerēšanu.

Lai šablonu drīkstētu izmantot noteiktam tipu komplektam, dotajos tipos ir jābūt definētām

visām darbībām, kas pielietotas šablona definēšanā.

template <typename T>

T add (T a, T b) {

return a + b;

};

**Parametru dinamisms ir iespēja veidot funkciju ar mainīgu parametru skaitu – variādisku funkciju.** Python ir iespēja norādīt funkciju parametrus netieši, un to iespējams organizēt divos veidos:

* **Parametru sarakstu** apzīmē viens speciālais parametrs, pirms kura nosaukuma ir viena zvaigznīte, un funkcija caur to padotos argumentus saņem kā slēgto sarakstu (tuple).
* parametru vārdnīca (tiks padoti argumenti ar nosaukumiem, apzīmē ar \*\*).

def printall(\*aa):

for a in aa:

print(a)

printall(1,2,3,4) # 1 2 3 4

def list\_lengths(\*\*kwargs):

for (key, values) in kwargs.items():

print(key, len(values))

list\_lengths(a=[1, 2, 3], b=[4, 5]) # {'a': 3, 'b': 2}

# Rekursija

**Rekursija ir process, kad process izsauc pats sevi. Funkcijas, kuras izsauc pašas sevi, sauc par rekursīvām funkcijām.**

Izstrādājot rekursīvu funkciju, jāatceras, ka tajā jāiekļauj nosacījuma operators, kurš nodrošina izeju no rekursīvā cikla. Ja tas netiks izpildīts, tad, vienreiz izsaucot funkciju, tā turpinās izsaukt sevi tik ilgi, kamēr notiks steka pārpildīšanās. Tas notiks tāpēc, ka funkcijas parametri un lokālie mainīgie glabājas stekā, un pie katra jauna izsaukuma stekā tiem tiek izdalīts atmiņas apgabals.

int fact (int num) {

if (num <= 1) return 1; // rekursijas bāzes bloks

else return num \* fact (num-1); // nodrošina rekursiju

}

# Masīvi C++ un to nodošana caur parametru

Masīvu nevar nodot caur funkcijas parametru kā vērtību (tādējādi veicot tā satura dublēšanu),

bet tā nodošana caur parametru iespējama tikai tādā veidā, kā tas notiek caur parametriem-referencēm.

* izmantojot masīvu notāciju – kvadrātiekavas []
* izmantojot norāžu notāciju – zvaigznīti \*

Kā citiem parametriem, pirms datu tipa var likt modifikatoru const, kas norādīs, ka masīvu funkcijā nedrīkst mainīt.

Var teikt, ka funkcijai netiek nodots pats masīvs, bet gan tikai norāde uz masīvu. Arī pats masīva mainīgais pēc būtības ir norāde. Sekojošās funkcijas ir izmantojamas kā statiskiem, tā dinamiskiem masīviem.

void function1 (char s[]) // ja padod simbolu virkni

void function2 (char \*s) // ja padod simbolu virkni

void function3 (int s[], int garums) // ja padod skaitļu masīvu

# Masīvi Python

Saraksts valodā Python atbilst masīva jēdzienam citās programmēšanas valodās. **Saraksts (list) ir indeksēta elementu virkne.** Vienu saraksta vienību sauc par elementu. Katru elementu identificē indekss – skaitlis, kas norāda, pie kura no elementiem vēršas lietotājs.

Izšķir šādus saraksta izveidošanas veidus:

* saraksta izveidošana ar inicializācijas virkni

aa = [1,2,3,4,5]

* saraksta izveidošana pēc cita saraksta

ee = aa # nav kopēšana!

* Kopēšana

bb = list(aa)

cc = aa[:]

dd = copy.deepcopy(aa) # visas dimensijas

* saraksta izveidošana ar ģeneratoru

aa = [(i+1)\*\*2 for i in range (10)]

Saraksta intervālu iegūst, izmantojot intervāla operatoru, tā tiek veidota intervāla kopija

print(aa[2:5]) # [a:b] no a līdz b neieskaitot

print(aa[1::2]) # katru otro, sākot ar pozīciju 1

print(aa[::-1]) # apgriezti

Saraksta izmēra izmaiņa par vienu elementu var notikt šādos veidos:

* append(elem) – elementa pievienošana beigās
* insert(pos, elem) – elementa pievienošana pēc pozīcijas
* elem = pop() – elementa izmešana no beigām (ar tā atgriešanu)
* elem = pop(pos) – elementa izmešana pēc pozīcijas (var būt negatīva)
* remove(val) – elementa (pirmā sastaptā) izmešana pēc vērtības
* del arr[i] – elementa dzēšana

Valodā Python ir vēl viens saraksta veids – slēgtais saraksts (tuple).

# Vairākdimensiju masīvi C++ un to nodošana caur parametru

**-dimensiju masīvā** katrs elements ir -dimensijas masīvs.

Viendimensijas masīva un vairākdimensiju masīvu padošana caur parametru sintaktiski neatšķiras, jo masīva mainīgā būtība ir vienāda abos gadījumos – norāde uz masīva sākumu. Līdzība ar viendimensijas (statisko) masīvu nodošanai caur parametru statiskam vairākdimensiju masīvam ir tāda, ka pirmajai dimensijai izmērs nav jāuzrāda, tomēr visām pārējām dimensijām ir.

void function1 (char s[][4][2]) // statiska masīva gadījumā

void function2 (char (\*s)[4][2]) // statiska masīva gadījumā

void function3 (char \*\*s, int count) // dinamiska masīva gadījumā

void function3 (int \*\*\*n, int g1, int g2, int g3)

# Dinamiski masīvi C++

Dinamiski vairākdimensiju masīvu elementiem var piekļūt, izmantojot tradicionālo masīvu sintaksi, tomēr tie atmiņā atrodas dažādos apgabalos – vienā kopējā apgabalā atrodas tikai pirmā dimensija.

Dinamisku vairākdimensiju masīvu veido pa dimensijām, un tā izmantošana notiek šādā secībā:

* atmiņas izdalīšana dimensijām, secībā no rupjākās uz smalkāko,
* dinamiska masīva izmantošana, izmantojot parasto masīvu sintaksi,
* atmiņas atbrīvošana pa dimensijām, sākot no smalkākās un beidzot ar rupjāko.

Dinamisku vairākdimensiju masīva mainīgo deklarējot, jālieto tik daudz zvaigznīšu, cik masīvam ir dimensiju:

char \*\*darr; // divdimensiju masīva mainīgā deklarēšana

darr = new char\*[n1];

for (int i=0; i<n1; i++) {

darr[i] = new char[n2]

}

...

for (int i=0; i<n1; i++) {

delete[] darr[i];

}

delete[] darr;

Dinamiski masīvi C++ parasti tiek pielietoti tad, programmās kompilēšanas brīdī nav zināms, cik lielam masīvam būs jābūt. To deklarē kā norādi un norādei piešķir jaunu dinamiskās atmiņas apgabalu ar operatoru new. Programmas beigās, vai tad, kad masīvs vairs nav vajadzīgs, vajag atbrīvot dinamiski piešķirto vietu ar delete.

# Iebūvētais dinamiskais masīvs STL::vector valodā C++

Standarta šablonu bibliotēka (*standard template library*, STL) ir C++ standarta bibliotēka, kas sevī ietver ģenēriskas (dažādiem datu tipiem paredzētas) datu struktūras un algoritmus.

STL klase vector (vektors) realizē dinamisku masīvu ar daudzām papildus īpašībām. Klase vector ir viena no populārākajām STL konteineru klasēm. Tās izmantošanas ērtums balstās uz piekļuves iespēju elementu vērtībām, izmantojot indeksu. Tajā pašā laikā jāņem vērā, ka, lai nodrošinātu piekļuvi pēc indeksa, šajā struktūrā nepieciešami mehānismi, kas prasa papildus skaitļošanas un atmiņas resursus šī servisa nodrošināšanai. Nepieciešams iekļaut bibliotēku vector.

Klases vector objekta izveidošana un aizpildīšana iespējama šādos veidos:

* tukša vektora izveidošana,

vector<int> aa;

* vairāku elementu vektora ar fiksētu vērtību izveidošana,

vector<int> bb (5,99);

* inicializācijas virknes izmantošana aizpildīšanai (sākot ar C++11),

vector<int> cc = {11,22,33};

* cita vektora vai masīva pilna vai daļēja kopēšana.

vector<int> ee (cc.begin(), --cc.end());

vector<int> aa = {11,22,33};

for (int i=0; i<aa.size(); i++) // pārstaigāšana ar indeksu

{ cout<< aa[i] << " "; };

for (auto &a: aa) // for cikls intervālam

{ cout<< a << " "; };

for (auto i=aa.begin();i!=aa.end();++i) // pārstaigāšana ar iteratoru

{ cout<<\*i<<" "; };

for\_each (aa.begin(),aa.end(),funkcija); // pārstaigāšana ar foreach

cout<<"Pirmais " << aa.front() << endl;

cout<<"Beidzamais " << aa.back() << endl;

Elementa pievienošana vektoram iespējama šādi:

* palielinot vektora izmēru (resize),

aa.resize(7,-1); // ar noklusēto vērtību -1

* beigās (push\_back),

aa.push\_back(99);

* vidū, pirms elementa, uz kuru norāda iterators (insert).

aa.insert(aa.begin(),77);

Elementu dzēšana no vektora iespējama šādi:

* no beigām (pop\_back),

aa.pop\_back();

* pēc iteratora (erase) – ja vajag no sākuma, izmanto šo,

aa.erase(aa.begin()); // izdzēš pirmo

* samazina izmēru (resize).

aa.resize(2); // samazina līdz 2 vērtībām

# Simbola datu tips C++ (char). Darbības un vērtību pieraksts

Viena simbola (*character*) glabāšanu atmiņā nodrošina datu tips char. Tas aizņem vienu baitu un nodrošina 256 dažādu vērtību glabāšanu. char vērtībai atbilst simbola literālis – simbols, kas ielikts vienkāršajās pēdiņās.

No informācijas apstrādes viedokļa tips char ir tas pats int, tikai vienā baitā, jo atmiņā simbolus reprezentē to kodi. atšķirība no int tipa tipam char izpaužas tikai ievadē un izvadē, kad vērtība tiek interpretēta vai formatēta par simbolu.

Valodā C++, lai no simbola iegūtu tā kodu vai otrādi, nav jāpielieto nekādas funkcijas – sliktākajā gadījumā jāpiemēro tipu pārveidošana. char tipa vērtībām piemērojama tā pati aritmētika, kas int vērtībām, tāpat arī salīdzināšanas operatori.

char c = 'A';

int i = c;

cout << c << endl; // A

cout << i << endl; // 65

cout << (char)i << endl; // A

# Zema līmeņa simbolu virkne C++ char\*. Darbības

**Zema līmeņa (C stila) simbolu virkne ir tāds masīvs, kuru raksturo divas papildus īpašības:**

* + **datu tips, no kura sastāv masīvs, ir simbols (char),**
  + **vismaz viens no masīva simboliem ir t.s. simbolu virknes beigu simbols.**

Valodā C++ zema līmeņa simbolu virknes ir simbolu virkņu noklusētais variants. Ikviens teksta literālis, piemēram, "Hello, World!". Simbolu virknes beigu simbola (jeb nulles simbola, '\0' vai vienkārši vērtība 0) esamība ir obligāts nosacījums, lai simbolu secību varētu nosaukt par simbolu virkni. Tā kā zema līmeņa simbolu virkne satur papildus nulles simbolu, tad simbolu virknes, kas satur tekstu garumā n, glabāšanai nepieciešamā atmiņa ir n+1 vienības.

Simbolu virknes sākumam nav obligāti jāsakrīt ar masīva sākumu, masīvā var glabāt vairāk nekā vienu simbolu virkni.

Attēls, kurā ir teksts, rinda, fonts, ekrānuzņēmums

Apraksts ģenerēts automātiski

Simbolu virkni masīvā var ievadīt no klaviatūras, izmantojot operatoru >>. Tomēr šai metodei ir viens liels trūkums: operators >> ir formatētās ievades operators, kurš tukšumus un citus atdalītājsimbolus uzskata par informācijas beigu pazīmi. Zema līmeņa simbolu virknes aizpildīšanai lietojama objekta cin metode getline. Papildus funkcijas ar bibliotēku <string.h>, nevar veikt aritmētiskas darbības tāpat kā masīviem. Var izdrukāt līdz beigu simbolam.

strcpy (old, new);

strcat (s, t); // s = st

strlen(s);

cout << s;

# Augsta līmeņa simbolu virkne C++ string. Darbības

Augsta līmeņa simbolu virknes valodā C++ ir objekti, kas nodrošina teksta glabāšanu un apstrādi, rūpējoties par nepieciešamo atmiņas vietu virknes saglabāšanai un nodrošinot ērtu sintaksi dažādām darbībām. Augsta līmeņa simbolu virkne izmanto bibliotēku <string>, bet ir iekļauta arī <iostream> un tās tips ir string.

Ievade no klaviatūras, izmantojot formatētās ievades operatoru, noved pie tām pašām problēmām – tukšumi un citi atdalītājsimboli tiek uzskatīti par beigu pazīmi. Tāpēc jāizmanto funkcija getline, kurai augsta līmeņa simbolu virknes gadījumā ir cita sintakse. Funkcija getline no faila (klaviatūras) lasa simbolus simbolu virknē, kamēr vien nav sastapts simbols ‘\n’ (ievadīts **ENTER**).

Informācijas kopēšana no vienas simbolu virknes uz otru, kā arī no zema līmeņa simbolu virknes, iespējama ar pierasto piešķiršanas operatoru, gan objekta deklarēšanas rindā, gan pēc tam. Nav jārūpējas ne par beigu simbolu, ne par to, vai pietiks vietas, kur virkni glabāt. Simbolu virknes pievienošana vienu otrai galā notiek, vai nu izmantojot operatoru +, vai nu piešķiršanas operatoru +=.

Augsta līmeņa simbolu virknes garumu var noskaidrot, izmantojot simbolu virknes objekta funkcijas size un length (strādā identiski).

Augsta līmeņa simbolu virknēs ir ērti glabāt un apstrādāt informāciju, tomēr daudzas standarta funkcijas pieprasa padot zema līmeņa funkcijas kā argumentus. Šim nolūkam der simbolu virknes objekta iekšēja funkcija c\_str, kas atgriež norādi uz augsta līmeņa simbolu virknei atbilstošo zema līmeņa virkni.

# Simbolu virkne Python

Simbolu virknes ir simbolu secības, kas nodrošinātas ar dažādām funkcijām ērtākai teksta apstrādei. Parastā Unicode simbolu virkne (str) automātiski pieņem un glabā simbolus jebkurā kodējumā, un šīs visas ērtības nodrošināšanai Python var tērēt pat daudzus baitus uz katru simbolu.

Simbolu virkne pieder pie nemaināmajiem datu tipiem (immutable), un to varētu tehniski salīdzināt ar slēgto sarakstu (tuple), sastāvošu tikai no simboliem. Simbolu virkni Python programmas tekstā parasti raksturo parastās vai dubultpēdiņas, kurās iekļauti simboli, nav atsevišķi izdalīts tāds datu tips kā viens simbols. Simbolu virknes maiņa it kā ir iespējama, bet patiesībā izveidojas jauns simbolu virknes objekts vecā objekta vietā.

s = 'Hello'

t = s # tā ir tā pati simbolu virkne, pieejama no cita mainīgā

print(t is s) # vai abi mainīgie norāda uz vienu objektu - jā

s += ', World!' # tagad s jau ir cita virkne, nevis pamainīta vecā

print(t is s) # vai abi mainīgie norāda uz vienu objektu – nē

i = str(1234) # cita objekta pārveidošana ar funkciju

z = '''Hello,

World!''' # vairākrindiņu teksts

ss = s.split() # saskalda pēc tukšumiem

t = ":".join(ss)

print(ord("A")) # burta kods

print(chr(66)) # burts pēc koda

Simbolu virknēm var pielietot konkatenāciju ar operatoru +, kā arī pavairot ar operatoru \*. Simbolu virkni saskalda apakšvirknēs ar split (apakšvirkņu robežas nosaka tukšuma simboli), un izveidojas saraksts no apakšvirknēm. Pretēja darbība notiek ar join, kas izveido simbolu virkni no saraksta ar virknēm.

# Klases vispārējā struktūra C++. Metožu realizācija ārpus klases

**Klase (class) ir objekta tipa apraksts, kas ietver sevī iekšējo mainīgo deklarēšanu un iekšējo funkciju aprakstu, kā arī pieejas tiesību definēšanu mainīgajiem un funkcijām.**

Obligāta sastāvdaļa klases definēšanai ir galvene. Klases galvenē tiek definēti visi lauki, tomēr klases metodes galvā var tikt uzrādītas divās detalizācijas pakāpēs:

* gan pilnībā
* gan var tikt norādīti tikai to prototipi, lai pēc tam metodi realizētu citur.

Klases metodes realizācija ārpus klases hedera ir gandrīz tāda pati kā parastas funkcijas realizācija, bet, tā kā metode nav neatkarīga funkcija, tad, atrodoties ārpus galvas, metodei ir jāpieliek papildus informācija – pie kuras klases tā pieder. To dara, izmantojot **redzamības atrisināšanas operatoru** ::

class simple {

int a;

public:

void set (int x);

void print ();

};

void simple::set(int x=1) {

a = x;

}

void simple::print() {

cout << a << endl;

}

# C++ konstruktors. Konstruktoru specifiskā sintakse

**Konstruktors ir klases iekšēja funkcija, kas tiek automātiski izsaukta pēc objekta izveidošanas.** Klasē var definēt vairākus konstruktorus ar atšķirīgām signatūrām. Konstruktora nosaukums sakrīt ar klases nosaukumu, tam nav atgriežamās vērtības tipa. Konstruktora pienākums ir nodrošināt korekta objekta izveidi.

class person {

int age;

public:

person (int age) {

if (age > 0 && age < 120) this->age = age;

else { cout << “Nederīgs vecums, iestatīts vecums 17.”;

this->age = 17; }

}

};

# Noklusētais konstruktors. Kopijas konstruktors

Katrā klasē ir vismaz viens konstruktors. Ja konstruktors nav definēts, tad kompilators automātiski izveido **noklusēto konstruktoru** bez parametriem ar tukšu ķermeni. Ja klasē ir definēts kaut viens konstruktors, tad noklusētais konstruktors automātiski netiek veidots un, ja ir nepieciešamība pēc konstruktora bez parametriem, tad tas obligāti jādefinē pašam programmētājam.

class person {

...

person (const char \*n, int a);

person () {}; // tukšais konstruktors, izveidots tieši

...

};

**Kopijas konstruktors** ir konstruktors ar vienu parametru, kur parametrs ir ar tipu reference vai konstanta reference uz šīs pašas klases objektu. Kopijas konstruktora būtība ir dot iespēju izveidot vienu objektu pēc cita objekta parauga.

class person {

...

person (const person &p) {

strcpy (name, p.name);

age = p.age;

}

};

# C++ destruktors. Noklusētais destruktors

**Destruktors ir specifiska klases iekšējā funkcija, kas automātiski tiek izsaukta tieši pirms objekta likvidēšanas.** Destruktora nosaukums sakrīt ar klases nosaukumu, bet tam priekšā ir tildes simbols. Destruktoram nav parametru, tādējādi klasei var būt tikai viens destruktors. Destruktoram ir jēga tikai tad, ja klase izmanto kaut kādus resursus, kuri, objektam likvidējoties, ir jāatbrīvo (piemēram, jāatbrīvo dinamiski izdalītā atmiņa vai jāaizver fails).

Destruktora izsaukšana atšķiras atkarībā no objekta izmantošanas veida.

* Ja objekts tiek izmantots tiešā veidā, tad destruktors tiek automātiski izsaukts tā bloka beigās, kad objekts deklarēts.
* Ja objekts ir dinamisks, tad destruktors tiek netiešā veidā izsaukts ar operatoru delete.

Noklusētais destruktors, līdzīgi kā noklusētais konstruktors, ir ar tukšu ķermeni, un tas neko nedara.

class person {

...

~person (){ // destruktors

delete[] arr;

cout << “likvidēts objekts” << this;

}

};

# Klases elementu redzamība. Klases statiskie un lokālie mainīgie

Klases redzamības apgabalā atrodošies mainīgie un funkcijas pēc pieejamības:

* publiskie jeb atklātie (public), kas ir pieejami arī no ārējiem moduļiem,
* privātie jeb slēptie (private), kas ir pieejami tikai no klases iekšējām funkcijām,
* aizsargātie (protected), kas pieejami arī no mantotajām (*inherited*) klasēm.

**Specifikatori** darbojas no pieminēšanas brīža vai nu līdz nākošajam specifikatoram, kas to atceļ, vai nu līdz klases galvas beigām, ja vairs neviena specifikatora nav. Turklāt no klases galvas sākuma, ja nav uzrādīts savādāk, pēc noklusēšanas darbojas private. Šie trīs specifikatori, lietoti klases galvā, sadala klases galvu sektoros, kur katrā sektorā ir noteikts vienots pieejamības līmenis.

Klases statiskie elementi (mainīgie un funkcijas) ir tādi klases elementi, kas ir kopīgi visiem klases objektiem neatkarīgi no to skaita. Vēl vairāk – tie eksistē arī tad, ja nav izveidots neviens klases objekts. Klases statiskos elementus parasti izmanto, lai izvairītos no globālajiem mainīgajiem (kas pēc būtības arī ir statiskie lauki) un ārpus klases funkcijām. Pie statiskajiem klases elementiem var vērsties gan tradicionālajā veidā, kādam klases objektam piemērojot attiecīgi operatorus . vai ->, gan neatkarīgi, pirms attiecīgā mainīgā vai funkcijas pievienojot klases vārdu ar redzamības atrisināšanas operatoru ::

Klases statisko elementu nosaka modifikatora static pievienošana pirms elementa klases galvā. Papildus tam, ka lauks klases galvā tiek deklarēts kā statisks, ārpus klases galvas tas vēl papildus ir jādefinē (uzreiz klases galvā to darīt nedrīkst).

class months {

static char monthnames[][10];

...

};

char months::monthnames[][10] = {"january", "february","march"};

# Rādītājs, adrese un reference

**Rādītājs ir mainīgais (vai cits mehānisms), kas glabā atmiņas adresi.** Šajā gadījumā var teikt, ka mainīgais reprezentē atmiņas apgabala sākumu, kurā glabāt datus. Norādes mainīgos parasti raksturo zvaigznīte \* pie to deklarēšanas, vienīgais izņēmums ir statiski masīvi, kas tiek deklarēti, izmantojot kvadrātiekavu notāciju, tomēr savā būtībā arī ir norādes, kaut arī ar ierobežotām izmantošanas iespējām.

Norāde savu vērtību, adresi, var iegūt šādos 3 veidos:

* statisks masīvs kā norāde – automātiski pie deklarēšanas,

char arr[20];

* iegūstot to no dinamiskās atmiņas piešķiršanas operatora vai citas norādes,

char\* arr2 = new char[20];

char\* arr3 = arr;

* noteiktam mainīgajam – izmantojot adreses ņemšanas operatoru &.

int i[20];

int\* pointer = &i[3];

pointer++; // pārvietojas pa 4 baitiem, jo int\*

Saistībā ar norādēm izšķir divas pretējas darbības:

* adreses iegūšana noteiktam atmiņas apgabalam pēc to reprezentējošā mainīgā, izmantojot adreses ņemšanas operatoru &
* vērtības iegūšana pēc adreses ar zvaigznītes operatoru \*

**Adrese** **ir vērtība, kas apzīmē noteiktu vietu atmiņā.** To iegūst, pieliekot adreses operatoru & pirms mainīgā. Adrese ir tehnisks jēdziens, kas domāts programmas iekšējām vajadzībām, tāpēc norāžu saturu tiešā veidā parasti nemēdz interpretēt.

**Reference:**

* **(tehniski) ir norādes paveids ar ierobežotām izmantošanas iespējām, bet ērtāku sintaksi.**
* **(no izmantošanas viedokļa) ir mainīgā sinonīms.**

Tipiskākais referenču pielietojums ir funkciju parametros, tomēr vispārīgā gadījumā to pielietojums var būt plašāks – un parasti tikai, lai uzlabotu programmas lasāmību, salīdzinot ar alternatīvo variantu, lietojot norādes. Referenci norāda references operators &. To nedrīkst sajaukt ar adreses ņemšanas operatoru – references operators piedalās mainīgā (vai parametra) deklarēšanā. Vēl viena references atšķirība no norādes ir tāda, ka references mainīgo nevar deklarēt bez vērtības piešķiršanas tam deklarēšanas brīdī. Referenci lieto arī funkcijas atgriežamajā vērtībā, tādējādi nodrošinot, ka funkcijas izsaukums var būt piešķiršanas operatora kreisajā pusē.

int a = 999;

int &ra = a; // reference (sinonīms) uz a

int \*pa = &ra; // mainīgā a adrese

(\*pa)++; // a vērtība tiek palielināta caur adresi pa

cout << a << endl; // 1000

# Formālie programmēšanas valodas sintakses definēšanas līdzekļi

**ISO metavaloda ir teksta formas apraksta metode, kas papildināta ar dažām teksta formatēšanas iespējām (šrifts, apakšraksts).** Pieraksts metavalodā dod iespēju par konkrētu konstrukciju noskaidrot, vai šī konstrukcija ir pareiza. Sintakses apraksta metode sastāv no divu veidu elementiem:

* terminālie simboli, t.i., tie, kas tiešā veidā parādās programmas tekstā, un
* neterminālie simboli, jeb jēdzieni, kas, izmantojot aprakstu, tiek noteiktā veidā izvesti, lai beigu beigās nonāktu līdz terminālajiem simboliem (kursīvs)

Bez elementiem sintaktiskajā pierakstā tiek izmantotas arī sintaktiskā pieraksta operācijas, kas nosaka, kādā secībā elementi tiek izkārtoti, lai atbilstu noteiktam neterminālajam simbolam:

* izvēle starp variantiem (one of)
* neobligātais statuss (optional)

*integer*:

*signopt* *unsigned\_integer*

*unsigned\_integer*:

*digit*

*digit* *unsigned\_integer*

*sign*:

+

-

*digit*: one of

0 1 2 3 4 5 6 7 8 9

# UML klašu diagrammas

**Objektorientētā modelēšana (OOM)** ir pieeja lietojumprogrammas modelēšanai, kas tiek izmantota programmatūras dzīves cikla sākumā, ja programmatūras izstrādei tiek izmantota objektorientēta pieeja.

**Vienotā modelēšanas valoda (UML)** ir vispārējas nozīmes vizuālās modelēšanas valoda, kas paredzēta, lai nodrošinātu standarta veidu, kā vizualizēt sistēmas dizainu.

**UML klašu diagrammas** ir diagrammas, kuras apraksta sistēmas struktūru, parādot klases, citus atribūtus, metodes un saistības starp objektiem.

UML klašu diagrammā klases tiek parādītas kā kastītes ar trim nodalījumiem: 1) nodalījums satur klases vārdu, 2) nodalījums satur klases atribūtus un 3) nodalījums satur klases metodes. Datu redzamībai šādās diagrammās arī ir dažādi apzīmējumi. Publiskos datus atzīmē ar +, privātos ar – un aizsargātos ar #

