Funktsioonid ja funktsioonide teegid

Mõisted: Alamprogrammid: Protseduurid.ja funktsioonid. Parameetrid. Lokaalsed ja globaalsed muutujad. Funktsioonide teegid.. Ülalt-alla projekteerimine ja struktuurne dekompositsioon.

Alamprogramm (Subprogram)

> **Alamprogramm** on osa kõrgkeeles kirjutatud programmist, mis teeb ära kindlalt ja loogiliselt piiritletud, kogu programmi jaoks vajaliku töö.

Mõistetena kasutatakse eesti keeles paralleelselt **alamprogrammi** ja **protseduuri** ja inglise keeles *procedure*, *subprogram* ja *subroutine*. Kuna protseduuride kasutamine on oluline võte veidigi suuremate programmide kirjutamisel, nimetatakse protseduure lubavaid kõrgkeeli **protseduurseteks programmeerimiskeelteks** .

Programmeerimise üsna varajases etapis märgati, et tihti on vaja programmis sama tegevust korrata mitmes kohas ja sellest kasvas välja idee korratav tegevus üks kord kodeerida ja mitmes kohas käivitada. Klassikaline näide on matemaatilised tegevused: nt ruutjuure või siinuse leidmine.

On olemas **standardprotseduurid** (sisseehitatud pr, protseduuride teegid) - *(intrinsic, built-in, library)*, mis on programmeerimiskeele koosseisus.

Ja on programmeerija enda kirjutatud protseduurid, st programmeerimiskeeles on olemas vahendid, et programmeerija saaks ise protseduure-funktsioone kirjutada ja hiljem kasutada neid sarnaselt standardprotseduuridega. Viimane on vajalik seetõttu, et keeled on ebaühtlaselt varustatud valmis alamprogrammidega ja põhimõtteliselt on võimatu ettenäha kõiki tarvilikke alamprogramme.

Protseduuri käivitamiseks *(invoke)* tuleb ta programmi sees **väljakutsuda** *(call)*. Tavaliselt tähendab väljakutse seda, et programmi sisse kirjutatakse protseduuri nimi ja lisatakse talle vajalikud parameetrid.

Alamprogramme saab nende käitumise järgi jagada kahte klassi (sõltub keelest, kui eristatavad nad on):

protseduurid - teevad midagi kasulikku ja tema väljakutsega võib asendada osa programmist

funktsioonid - arvutavad ühe väärtuse, mida saab kohe edasi kasutada avaldises

Kuidas konkreetses keeles midagi vormistatake, on väga erinev, ka see, kas on eristatavad protseduurid ja funktsioonid. Erinevuseks on ka see, kas alamprogrammid tuleb vormistada peaprogrammi osadena (keele nn **plokk-struktuur**) või on neid võimalik esitada ka füüsiliselt eraldi (eraldi failidena).

C-s on vaid üks variant alamprogrammi kirjutamiseks. Täpsemalt öeldakse, et C-keeles on kõik, isegi peaprogramm funktsioon.

Plokk-struktuuriga keel *(block structure)*

Mõiste ja kontseptsioon programmeerimiskeeltes, mis lubab omavahel seotud deklaratsioone ja lauseid gruppeerida. Selle võimaluse mõistliku kasutmise korral saab suure programmi muuta hästi struktuurseks ja arusaadavaks. Plokk-struktuur on paljudes peale 1960 aastat loodud protseuursetes keeletes mingil kujul olemas, alguse sai ta ALGOL-60st.

Programmeerija jaoks on plokkstruktuuril kaks funktsiooni:

1. Lubab täidetavad laused gruppeerida nn **liitlausetesse** *(compound statement)* - saab teha liitlause igale poole, kus keel lubab kasutada ühte lauset ja nii saab mõelda lausete jadast kui

ühest tegevusest, muutes programmi loomise lihtsamaks. Plokk võib sisaldada teisi plokke ja tekib struktuur, kus plokid on üksteise sees. Programmi loogika on ülesehitatav hierarhiliselt, plokkide kaupa.

2. Juhib **muutujatele mälu eraldamist** ja muutujate kättesaamist. See võimaldab programmi täitmise jooksul dünaamiliselt hõivata ja vabastada mälu. Ploki sees deklareeritud muutujatele eraldatakse mälu plokki sisenemisel ja (kui pole eraldi midagi tehtud), siis vabastatakse taas plokist väljumisel. Iga plokk saab olla muutuja **skoobiks** e kehtivuspiirkonnaks. Muutuja kehtib vaid selles plokis ja tema sees olevates plokkides. Oluline point on see, et need muutujad ei oma mõju väljapoole ja neid ei saa muuta plokist väljaspool. Selline süsteem loob võimaluse **info peitmiseks** *(information hiding)* - andmeid ei saa muuta ja neile ei pääse ligi väljaspoolt plokki.

C-keel on kirjeldatud mõttes plokkstruktuuriga keel. Plokke eristatakse looksulge {} kasutades.

Plokk-struktuuriga programmides näeb plokk välja tavalislet järgmine

algus

<ploki päis>

<ploki keha>

lõpp

Ploki päis koosneb muutujate ja alamprogrammide deklaratsioonidest (võib ka olla tühi); ploki keha koosneb täidetavatest lausetest.

Plokk saab koosneda täidetavatest lausetest ja plokk on ise täidetav lause. Plokke saab panna üksteise sisse suvalisele sügavusele.

Sellest tuleneb järgnev:

1. Sama identifikaatorit saab kasutada **ploki sees vaid ühe objekti jaoks**, kuid sõltumatutes plokkides saab kasutada samu identifikaatoreid erinevate objektide jaoks. Hea programmerija ei pinguta selle võimalusega üle - mõistlik on kasutada samu identifikaatoreid tsüklimuutujateks, loendamiseks, kuid mitte olulismate andmete meelespidamiseks, mis programmist arusaamist ohustab.
2. Sama identifikaatorit saab kasutada **erinevates üksteise sees olevates plokkides** erinevate objektide tähistamiseks. Segaduste vältimiseks kasutatakse järgmisi skoobi reegeid (pärit algselt keelest ALGOL). Leitud muutuja deklaratsiooni hakatakse otsima kõigepealt sama ploki päisest, siis selle ploki päisest, milles antud plokk paikneb jne kuni peaprogrammini välja. Kõige esimesena ette jääv (sisemisem) deklaratsioon on see, mis loetakse kehtivaks.
3. **Uude plokki sisenemisel eraldatakse mälu** plokis deklareeritud muutujatele (seda mälu nimetatakse ploki **aktiveerimiskirjeks** *(activation record)*), kui plokist lahkutakse, vabastatakse mälu. Plokkidest väljumisel ja mälu vabastamisel kehtib LIFO põhimõte, st pinu mehhanism on sobiv aktiveerimiskirjete salvestamiseks. Mälu, mis viimati reserveeriti, vabastatakse esimesena.

Konkreetse keele puhul on vaja välja selgitada, kuidas on plokk määratud.

Globaalsed ja lokaalsed muutujad

Kõiki programmis kasutusel olevaid muutujaid ei pruugi saada kasutada (lugeda, muuta) kogu programmi seest, vaid on reeglid (tavaliselt sõltuvalt keelest), millise piirkonnas midagi kättesaadav on. Lausete osa programmis, kus muutuja on kättesaadav, kutsutakse selle muutuja **kehtivuspiirkonnaks** e **skoobiks** *(scope)*. Plokk-struktuuriga keele puhul on selleks piirkonnaks

tavaliselt plokk, kus muutuja on deklareeritud, samuti need plokid, mis antud ploki sisse jäävad.

Muutuja, mis on deklareeritud antud ploki sees, on selle ploki jaoks **lokaalne muutuja** *(local variable)*. Kui muutuja on deklareeritud nö kõrgemas plokis, on ta kõrgema ploki sees olevate plokkide jaoks **globaalne muutuja** *(global variable)*. Lisaks on tavaliselt nii, et kui seesmises plokis peaks kehtima globaalne muutuja välimisest plokist ja samas seesmises plokis on deklareeritud sama nimega lokaalne muutuja, siis reaalselt kehtib ja on kasutatav lokaalne muutuja (sama nimega muutujate puhul on lokaalne "tähtsam"). Kirjeldatud süsteem tuleneb otse eespool kirjeldatud muutujadeklaratsioonide otsimise strateegiast, kus sellega alustatakse samast plokist, kus muutuja "avastatakse".

Globaalsete muutujate kasutamist AP-de sees loetakse halvaks stiiliks, sest sel teel võib kergesti tekkida raskelt avastatavaid vigu. Saab tekkida nn **kõrvalefekt** *(side effect)* - so globaalse muutuja muutmine alamprogrammi seest. Enamasti ei ole globaalse muutuja muutmine planeeritud ja seega mittesoovitatav. Vähegi suurema koodi korral on väga oluline osata ennustada, kuhu ja millisel viisil mõjuvad mingis koodi osas tehtud muudatused. Selles kontekstis on ka oluline selge ja läbimõeldud alamprogrammide kasutamine koos sisendandmete ja väljundiga.

Seda ei saa muidugi 100% nii võtta - kui programm tegeleb näiteks andmebaasiga, siis viimane on globaalne kõigi alamprogrammide jaoks ja tema muutmine ja temaga tegelemine ongi kogu eesmärk.

Argument või parameeter

Parameetrid on mõistlik vahend andmete edastamiseks alamprogrammidesse

> **Parameeter** on väärtus, mis antakse protseduurile või funktsioonile ette, et ta saaks välja arvutada väärtuse või teha mingi muu olululise (parameetrist sõltuva) tegevuse (nt sqrt(arv) - arv on parameeter).

Eristatakse:

formaalseid *(formal, dummy)* **paremeetreid** ja

tegelikke *(actual, calling)* **parameetreid**.

Formaalseid parameetreid kasutatakse alaprogrammi päises tema kirjeldamiseks, **tegelikke** aga AP väljakutses, et AP-le tegelikke väärtuseid ette anda. Formaalsete parameetritena tuleb alati kasutada muutujaid, tegelikeks parameetriteks võivad olla ka konstandid ja avaldised.

Edasi sõltub palju konkreetsest keelest, millisel viisil ja kui rangelt parameetrite "maailm" korrastatud on. Võib kehtida reegel, et formaalsed ja tegelikud parameetrid on sama tüüpi (üsna tüüpiline), parameetrid võivad olla jagatud sisend- ja väljundparameetriteks. Esimesed annavad ette lähteandmed ja teiste kaudu saadakse teada vastused. Võib olla kehtestatud, et formaalseid ja tegelikke parameetreid peab olema sama arv. Kuid võivad olla ka võimalused määrata parameetitele vaikeväärtuseid, jätta osa tegelikke parameetreid ära jms.

Parameetrite edastamine (Parameter passing)

Kuidas toimub parameetrite edastamine alamprogrammi?

Eristatakse kolme tehnikat:

1. **Väärtuskutse** *(Call by value)* - AP-le antakse ette väärtus ja pole mingit võimalust jõuda tagasi väljakutsuva programmi elementide/muutujate juurde. Näiteks saab olla tegelikuks parameetriks X*Y ja korrutise väärtus saab väärtuseks formaalsele parameetrile funktsiooni sees. AP ei saa muuta tegelikuks parameetriks olevate muutujate väärtuseid. Avaldise või

konstandi kasutamisel argumendiks siseneb AP-i ainult väärtus.

2. **Aadresskutse** *(Call by reference)* - AP-le ei edastata tegeliku parameetri väärtust, vaid mäluaadress, kus see paikneb. Edasi on AP vastutusel, kuidas ta parameetritega käitub AP jagab teda väljakutsunud programmiga sama mälu. Sellise edastusviisi puhul pole tavaliselt (Näiteks Pascali) lubatud parameetritena kasutada ei konstante ega avaldisi.

On kaks juhust, kus variandid 1 ja 2 käituvad erinevalt:

- kui parameetriks on massiiv, siis väärtuse järgi *(by value)* edastamiseks tuleb ka AP-s eraldada mälu näiteks 1000 elemendi jaoks, kuhu tehakse massiivi koopia. Kui aga kutsutakse aadressi *(by reference)* järgi, piisab esimese elemendi aadressi edastamisest AP-le.

- kui formaalne parameeter on AP-s omistuslause vasakul poolel, siis väärtuse järgi väljakutse korral muutub argumendi väärtus ainult alamprogrammis (sisuliselt on tegemist sisendparameetriga). Kui on aga väljakutsumine aadressi järgi, muutub väärtus ka väljakutsuvas programmis (tegu on väljundparameetriga). Kui eesmärgiks ongi AP abil argumendi väärtust muuta, on saavutatud see, mida vaja. Vastasel juhul saame aga soovimatu kõrvalefekti.

3. **Nimikutse** *(Call by name)* - tegelik avaldis ise saadetakse AP-i, mitte stringina, vaid väikese masinkoodid programmijupikesena. Lisaks koodile, mis tuleb välja arvutada, sisaldab see jupike ka viiteid muutujatele, muutuja keskkond saab aga muutuda. Seega iga kord, kui väljakutsutud AP-s viidatakse parameeter X-le, arvutatakse ta koodijupikest kasutades uuesti välja. Nii võib tekkida olukord, et X-i väärtus AP-i jooksul muutub. Kui argumendiks on lihtsalt muutuja, pole vahet võrreldes aadressi järgi väljakutsumisega. Vahe tekib siis, kui on tegemist avaldisega.

C-keeles kasutatakse parameetrite jaoks väärtuskutset (call by value).

Et aga keeled erinevad, peab programmeerija tutvuma konkreetse keele eripäradega, et soovimatuid kõrvalefekte vältida.

Suuremate programmide koostamiseks on hea, kui alamprogramme saab eraldi failidesse paigutada (AP ja PP võivad paikneda füüsiliselt eraldi).

Moodul (modul) või teek (library)

Süsteemi või programmi osa saab defineerida kui loogiliselr eraldatud osa, mis ise sisaldab kõiki enda jaoks vajalikke definitsioone - nagu andmetüüpide ja muutujate kirjeldused jms. Erinevates keeltes kutsutakse selliseid eraldi failis paiknevaid osai erinevalt. C-keel kasutab mõistet **teek** (*library*) Mitmetes keeltes on aga kasutusel sõna **moodul.** Hästi koostatud moodulis on selge sisu ja struktuur ning hästi kirjeldatud tegevused. Lisaks on korralikult defineeritud võimalikud sisendid ning tagastatavad väljundid.

Hea programmi projekteerimine algab programmi funktsionaalsuse üldisemast kirjeldusest ja liigub mööda suurenevat täpsustamist detailsuse poole. Sellist tehnikat kutsutakse *top-down design*.

Moodulite ülesehitused keeleti on samuti erinevad.

Informatsiooni peitmine (information hiding)

Kolm olulist probleemi tarkvarasüsteemide loomisel on:

1. Töö jaotamine väiksmateks osadeks selliselt, et programmi kirjutamist oleks võimalik jagada programmeerijate vahel
2. Valmidus muudatuste sisseviimiseks

3. Keerukuse haldamine

Tööd on mugav programmeerijatele kätte jagada moodulite kaupa ja informatsiooni peitmine on oluline printsiip programmi moodulitesse jaotamisel. Iga moodul peab olema sõltumatult väljatöötatav ja ka muudetav ning mõistetav.

Info peitmise printsiip ütleb, et need andmed/suurused/väärtused, mis moodulist välja tulema ei pea, peab olema moodulisse ära peidetud (kasutatav mõiste on **kapseldamine** - *encapsulaton* - mida kiputakse seostama vaid OOP-iga, kuid päris nii see pole). Neid ei tohi kätte saada, kuid veel vähem võib neid muuta.

Moodulisse peidetakse, milliste andmestruktuuride ja algoritmide abil töö on korraldatud. Mooduli programmeerija ülesandeks on anda mooduli kasutajale teatud võimalused mooduli kasutamisel, samal ajal mitte reetes saladust e kuidas need võimalused kasutajani jõuavad. Seega moodul peab andma liidese välisele kasutajale (kust viimane saab teada kõik enda jaoks vajaliku) ja peitma ära realisatsiooni, et see kasutajat ei segaks ega ahvatleks midagi seal sees ringi tegema.

Info peitmine annab võimaluse sõltumatult muuta mooduli realisatsiooni ning moodulit sõltumatult kontrollida ja mõista. Põhistruktuur peab olema mõistetav ilma realisatsiooni detalideta.

Mooduli liides peab sisaldama kogu vajalikku informatsiooni mooduli kasutajale. See peab sisaldama nii mooduli teenuste (tegevuste, alamprogrammide) kasutamise süntaksi kui ka semantikat (mis juhtub, kui mooduli teenust kasutada).

Tarkvaraarenduse meetodite arendamine on seotud info peitmise põhimõtte kasutamisega. On aluseks OOP meetodile, kus tarkvara on organiseeritud objektideks ja nad on üksteise realisatsioonidest sõltumatud. Samuti on info peitmine oluline põhimõte andmete organiseerimisel abstraktsetesse andmetüüpidesse. Sel puhul on saladuseks see, kuidas andmetüüp täpselt realiseeritud on ning milline operatsioonide realisatsioon.

On keeli, mis lasevad mooduleid üles ehitada kahes osas - liidesena ja realisatsioonina. Kuulsamad keeled selles osas on ADA oma package'iga, Modula-2 moodulitega.

C-s saab teegist eraldada nn *header*-faili, mis jääb tekstifaili kujule ja mida saab lisada include-käsuga programmi teksti juurde. Teek ise funktsioonide koodidega on kasutamiseks kompileeritud kujul.