თემა 6. ქვეპროგრამები

* ქვეპროგრამები – პროცედურები და ფუნქციები
* ქვეპროგრამების გამოძახება
* პარამეტრები
* ქვეპროგრამისთვის პარამეტრების გადაცემის ხერხები

სემანტიკა copy-in ან გამოძახება მნიშვნელობით call-by-value

სემანტიკა copy-out

„მითითების სემანტიკა“ (reference cemantics) ან „გამოძახება მითითებით“ (call-by-reference)

**ქვეპროგრამები – პროცედურები და ფუნქციები**

ქვეპროგრამა არის პროგრამული კოდის ცალკეული ერთეული (სეგმენტი), რომელზეც შესაძლებელია არაერთხელ მიმართვა პროგრამის ნებისმიერი ადგილიდან. ქვეპროგრამების გამოყენებას აქვს რამდენიმე მიზეზი:

* პროგრამის ფრაგმენტი, რომელიც უნდა შესრულდეს პროგრამის მუშაობის სხვადასხვა ეტაპზე, შეიძლება გაფორმდეს ცალკე, შემდეგ კი შესრულდეს იქ, სადაც საჭიროა და იმდენჯერ, რამდენჯერაც საჭიროა (კოდის განმეორებადი გამოყენება – reusability). მეხსიერების ეკონომიის გარდა, ეს მიდგომა აგრეთვე გვიცავს კოდის კოპირების შესაძლო შეცდომებისაგან.
* ქვეპროგრამის გამოყენება გვაძლევს საშუალებას დიდი ამოცანა დავყოთ პატარა ქვეამოცანებად (პროგრამის დეკომპოზიციის განხორციელება), რაც აიოლებს დაპროგრამების პროცესს. ამ დროს თითოეული ამოცანის ალგორითმი ფორმდება ქვეპროგრამის სახით. ე.ი. ქვეპროგრამა – ცალკეული ამოცანის ამოსახსნელად განკუთვნილი და გარკვეული წესით გაფორმებული ინსტრუქციების ჯგუფია.
* ქვეპროგრამის გამოყენება ამარტივებს პროგრამაში ცვლილებების შეტანას: საკმარისია შევიტანოთ ცვლილებები ერთხელ – ქვეპროგრამაში.
* რადგან ქვეპროგრამა პროგრამის დეკომპოზიციის ლოგიკური ერთეულია, მიზანშეწონილად ითვლება პროგრამის ნაწილის ქვეპროგრამად გაფორმება მაშინაც, როდესაც იგი მხოლოდ ერთხელ უნდა შესრულდეს. მიზანი – ტესტირების და დოკუმენტირების გამარტივება, პროგრამის წაკითხვადობის გაუმჯობესება.
* ქვეპროგრამა შეიძლება გამოყენებულ იქნეს ასევე როგორც დეკომპოზიციის ფიზიკური ერთეული, ე.ი. ცალკე კომპილირებადი ერთეული. მაგალითად, Fortran –ის ქვეპროგრამა (subroutine) არის როგორც დეკომპოზიციის ლოგიკური ერთეული, ისე კომპილირების ერთეული. თანამედროვე ენებში დეკომპოზიციის ფიზიკური ერთეულია მოდული, რომელიც წარმოადგენს განაცხადებისა და ქვეპროგრამების ჯგუფს.

ქვეპროგრამის ინტერფეისი შედგება

* მისი სახელისგან, პარამეტრების სიისაგან (თუ პარამეტრები არსებობს) და დასაბრუნებელი მნიშვნელობის ტიპისაგან.
* ლოკალური განაცხადებისაგან, რომლებიც მოქმედებენ მხოლოდ ქვეპროგრამაში
* შესრულებადი ინსტრუქციების თანმიმდევრობისგან

ლოკალური განაცხადები და შესრულებადი ინსტრუქციები შეადგენენ ქვეპროგრამის ტანს.

ქვეპროგრამებს, რომლებიც აბრუნებენ მნიშვნელობებას, ეწოდებათ *ფუნქცია* *(function)*, ხოლო რომლებიც არ აბრუნებენ – *პროცედურა (procedure)*.

მაგალითად, С –ს სინტაქსით პროცედურები გათვალისწინებული არ არის; ამის ნაცვლად ენაში გამოიყენება ფუნქცია დასაბრუნებელი void ტიპის მნიშვნელობით. ე.ი. С –ში ფუნქციის განმარტების ფორმა შემდეგია:

<ტიპი> <დასახელება> (<პარამეტრების სია>) /\* ეს არის ფუნქციის სათაური\*/

{

ფუნქციის ტანი

ლოკალური განაცხადები (თუ საჭიროა)

შესრულებადი ინსტრუქციების თანმიმდევრობა

}

სადაც

<ტიპი> განმარტავს ფუნქციის დასაბრუნებელი მნიშვნელობის ტიპს (მარტივი ან რთული ტიპი, პოინტერი მარტივ ან რთულ ტიპზე...) ან void (ტიპი, რომელსაც მნიშვნელობა არაა აქვს);

<დასახელება> – ფუნქციის სახელია. ფუნქციის სახელი წარმოადგენს იდენტიფიკატორს, რომლის საშუალებითაც კეთდება განაცხადი ფუნქციაზე (მოიცემა პროტოტიპი), ფუნქციის იმპლემენტაცია (შესასრულებელი ამოცანის ალგორითმის აღწერა) და ფუნქციაზე მიმართვა (ფუნქციის გამოძახება);

<პარამეტრების სია> წარმოადგენს ფუნქციის ფორმალური პარამეტრების ჩამონათვალს მათი ტიპების მითითებით. ჩამონათვალში პარამეტრები გამოყოფილია მძიმით, ამასთან ტიპი უნდა მიეთითებოდეს ყოველ პარამეტრს. პარამეტრების საშუალებით ფუნქციას პროგრამიდან გადაეცემა მისთვის საჭირო საწყისი მონაცემები;

<ფუნქციის ტანი> შეიცავს ამოსახსნელი ამოცანის რეალიზებისთვის საჭირო ცვლადებზე განაცხადებს და ინსტრუქციების ნაკრებს.

**ფუნქცების გამოძახება**

ვთქვათ, ამოცანაში, სხვა დანარჩენ მოქმედებებთან ერთად, დაგვჭირდება რამდენიმეჯერ 2 მთელ რიცხვს შორის უდიდესის დადგენა. С –ს შესაბამისი ფუნქციის სახია:

1.

int udidesi(int x, int y)

{

int m;

if (x > y) m = x;

else m = y;

return m;

}

2.

int udidesi(int x, int y)

{

if (x > y) return x;

return y;

}

3.

int udidesi(int x, int y)

{

return (x > y) ? x : y;

}

ხოლო გამოძახებას აქვს სახე:

int a, b;

scanf("%d%d", &a, &b);

int max = udidesi(a, b);

ან გამოსახულებაში

y = x + func(a, b) + 3;

Pascal –სა და Ada –ში ქვეპროგრამები ასევე რეალიზებულია ორი სახით: პროცედურები და ფუნქციები. ორი რიცხვის საშუალო არითმეტიკულის გამოთვლის ფუნქცია Pascal –ზე:

Function Average(A,B:integer) : real;

Begin

Average := (A + B)/2;

End;

გამოძახება:

Rez :real;

A, B : integer;

readln(A,B);  
Rez := Average(A,B);

**პროცედურების გამოძახება**

С –ში პროცედურა პროტოტიპით

void proc(int a, float b);

გამოიძახება როგორც

proc(x, y);

სადაც x მთელი რიცხვია, y – ნამდვილია.

Fortran ენაში მიმართვა პროცედურაზე შეიცავს ოპერატორს call:

call proc(x,y)

Pascal –ის პროცედურა და მისი გამოძახება:

1.

procedure subprog;

begin

WriteLn('Bye');

end;

begin

WriteLn('Hello');

subprog;

subprog;

end.

2.

procedure subprog(Line: String);

begin

WriteLn(Line);

end;

begin

WriteLn('Hello');

subprog('Good bye,');

subprog('my love,');

subprog('good bye!');

end.

С# –ის ფუნქცია

static int MaxValue (int [ ] intArray)

{

int maxVal = intArray[0];

for (int i = 1; i < intArray.Length; i++)

{

if (intArray[i] > maxVal)

maxVal = intArray[i];

}

return maxVal;

}

და მისი გამოძახება

static void Main(string [ ] args)

{

int[] myArray = {1, 8, 3, 6, 2, 5, 9, 3, 0, 2};

int maxVal = MaxValue(myArray) ;

Console.WriteLine("The maximum value in myArray is {0}", maxVal);

Console.ReadKey ();

}

ქვეპროგრამის გააქტიურება (ანუ მასში დაპროექტებული ამოცანის შესრულების დაწყება) ხდება ქვეპროგრამის გამოძახების გზით. ქვეპროგრამის გამოძახების სემანტიკა შემდეგია: როდესაც პროგრამას ხვდება ქვეპროგრამის გამოძახების ჩანაწერი, ინსტრუქციების მიმდინარე თანმიმდევრობის შესრულება შეჩერდება და მართვა გადაეცემა ქვეპროგრამას; შესრულდება მისი ტანის ინსტრუქციების ჯგუფი; შემდეგ მართვა უბრუნდება პროგრამას ქვეპროგრამის გამოძახების წერტილში; უფრო ზუსტად, პროგრამის შესრულება გაგრძელდება გამოძახების მომდევნო პირველივე ინსტრუქციიდან.

**პარამეტრები**

* ფორმალური პარამეტრები – ქვეპროგრამის ლოკალური ცვლადებია; მათზე განაცხადი კეთდება ქვეპროგრამის სათაურში, ხოლო გამოთვლები ქვეპროგრამის ტანში იწერება ფორმალური პარამეტრების ტერმინებში.
* ფაქტობრივი პარამეტრები (არგუმენტები) – ის მნიშვნელობებია, რომლებიც გამომძახი პროგრამიდან გადაეცემა ქვეპროგრამას.

int Sum(int x, int y); ფუნქციის პროტოტიპში x და y ქვეპროგრამა–ფუნქციის ფორმალური პარამეტრებია, ან, უბრალოდ , *პარამეტრებია*.

int a = 287, b = -143;

int answer = Sum (a, b);

აქ a და b ფუნქციის *არგუმენტებია*.

ზოგადად, ქვეპროგრამის გამოძახებისას არგუმენტების და პარამეტრების შესაბამისობა დგინდება პარამეტრის პოზიციის მიხედვით:

procedure Proc(First: Integer; Second: Character);

Proc(24, 'X');

ენა Ada –ს აქვს შესაძლებლობა ჩამოაყალიბოს პარამეტრების შესაბამისობა სახელის მიხედვით, ამასთან პარამეტრების რიგითობის დაცვა აუცილებელი არაა:

Proc(Second => 'X', First => 24);

როგორც წესი, ეს ვარიანტი და არგუმენტების ნაგულისხმევი მნიშვნელობები გამოიყენება ერთად:

procedure Proc(First: Integer := 0; Second: Character := '\*');

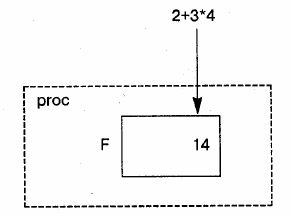
Proc(Second => 'X');

**ქვეპროგრამისთვის პარამეტრების გადაცემის ხერხები**

*სემანტიკა copy-in ან გამოძახება მნიშვნელობით call-by-value*

ეს არის არგუმენტის ქვეპროგრამაში გადაცემის წესი, რომლის დროსაც ხდება არგუმენტის ასლის ჩაწერა მეხსიერების იმ ადგილში, რომელიც გამოეყო ფორმალურ პარამეტრს. ამ მექანიზმს ეწოდება *„სემანტიკა* *copy-in*“ (კოპირება ქვეპროგრამაში) ან ქვეპროგრამის *„გამოძახება მნიშვნელობით“ (call-by-value).*

შემდეგი პროცედურისთვის (Ada)

procedure Proc(F: in Integer) is

begin

... ;

end Proc;

და მისი გამოძახებისთვის

Proc(2 + 3 \* 4);

ნაჩვენებია სემანტიკა copy-in:

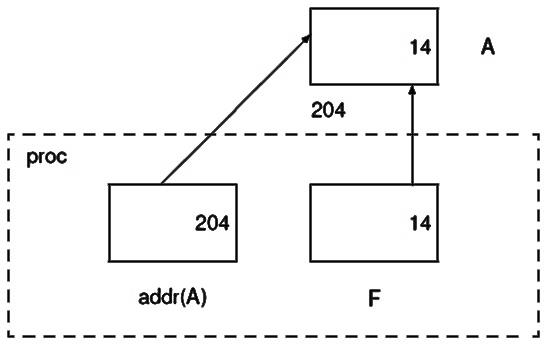
უპირატესობები:

* copy-in არის პარამეტრების გადაცემის ყველაზე სანდო მექანიზმი. რადგან ქვეპროგრამა ღებულობს არგუმენტის ასლს, მას არ შეუძლია არგუმენტის შეცვლა – ყველა მოქმედება ქვეპროგრამის შიგნით შესრულდება არგუმენტის ასლზე, და არა თვით არგუმენტზე.
* არგუმენტები შეიძლება იყოს მუდმივები, ცვლადები ან გამოსახულებები,
* copy-in მექანიზმი საკმაოდ ეფექტურია, ვინაიდან კოპირებასთან დაკავშირებული დანახარჯები კეთდება მხოლოდ ერთხელ, ხოლო ყველა დანაჩენი მიმართვა ფორმალურ პარამეტრზე სინამდვილეში წარმოადგენს მიმართვას ლოკალურ ასლზე და არის ძალზედ სწრაფი.

თუ კი copy-in სემანტიკა ამდენად კარგია, რატომღა არსებობს სხვა მექანიზმები? საქმე ისაა, რომ ხშირად ჩვენ გვჭირდება არგუმენტის შეცვლა ქვეპროგრამის მიერ.

* ქვეპროგრამა–ფუნქციას შეუძლია მხოლოს ერთადერთი მნიშვნელობის დაბრუნება პროგრამაში. ზოგჯერ კი შეიძლება საჭირო გახდეს ერთზე მეტი შედეგის დაბრუნება, და ჩნდება ისეთი პარამეტრების არსებობის აუცილებლობა, რომლებსაც შეეძლებათ „გაიტანონ“ ქვეპროგრამაში მიღებული შედეგები. თუმცა, ამას შეგვიძლია აუაროთ გვერდი, თუ დავაბრუნებინებთ ფუნქციას ჩანაწერის, სტრუქტურის ან კლასის ობიექტს.
* შესაძლოა ქვეპროგრამის შესრულების მიზანი სწორედ მასში გადაცემული მონაცემების მოდიფიცირება იყოს. მაგალითად, მასივის დალაგების ან შევსემის ქვეპროგრამა.
* დიდი არგუმენტის კოპირება არაეფექტურია ან ქვეპროგრამისთვის გამოყოფილი მეხსიერება მისი ასლისთვის საკმარისი არ არის.

პირველ ორ შემთხვევას ადვილად მოევლება *copy-out სემანტიკის* (კოპირება ქვეპროგრამიდან) გამოყენებით. მექანიზმი შემდეგია: არგუმენტი უნდა იყოს ცვლადი, ქვეპროგრამას გადაეცემა არგუმენტის მისამართი, რომელსაც იგი იმახსოვრებს დროებით ლოკალურ ცვლადში. ფორმალურ პარამეტრს ერთხელ მაინც უნდა მიენიჭოს გასატანი მნიშვნელობა, რომელიც ქვეპროგრამის მუშაობის დასრულების შემდეგ კოპირდება პროგრამის ცვლადში, რომელსაც მიუთითებს შენახული მისამართი.

შემდეგი პროცედურისთვის

procedure Proc(F: out Integer) is

begin

F := 2 + 3 \* 4;

end Proc;

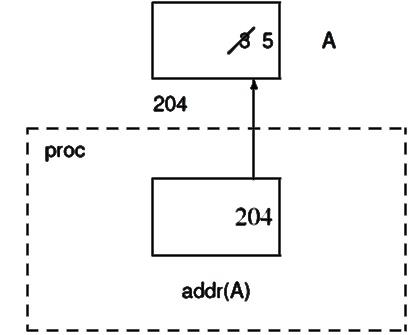
და მისი გამოძახებისთვის

A: Integer;

Proc(A);

სემანტიკა copy-out შემდეგია:

პარამეტრების გადაცემის მესამე ხერხი ცნობილია როგორც *„გამოძახება მითითებით“ (call-by-reference) ან „მითითების სემანტიკა“* (reference cemantics). მდგომარეობს არგუმენტის მისამართის გადაცემაში, ხოლო პარამეტრზე მიმართვა ხდება ირიბად (განმისამართების გამოყენებით).

შემდეგი პროცედურისთვის

**procedure** Proc(F: **in out** Integer) **is**

**begin**

F := F + 2;

**end** Proc;

და მისი გამოძახებისთვის

A: Integer := 3;

Proc(A);

*მითითების სემანტიკა* შემდეგია: