**ПАСКАЛЬ: точка повернення, підстановка аргументів, зберігання змінних.**

**1. Точка повернення**

*Місце в програмі, що описує дію, якою вона продовжується по закінченні виконання виклику підпрограми*, називається ***точкою повернення*** з підпрограми.

Отже, коли виконання програми доходить до виклику підпрограми, запам'ятовується точка повернення з неї, та *вказівка на цю точку зберігається до закінчення виконання виклику*. Потім за цією вказівкою продовжується виконання програми саме з точки повернення. Вказівка точки повернення займає частину ***локальної пам'яті процесу виконання виклику підпрограми***. Що ще знаходиться в цій пам'яті, ми скажемо трохи нижче.

Точка повернення з процедури – це позначення дій, виконуваних після її виклику. Це може бути наступний за її викликом оператор, як у програмі *sort31*. Або умова завершення чи продовження циклу, якщо виклик записаний останнім у тілі циклу.

Після виклику функції може застосовуватися операція до значення, що повертається з виклику, або обчислюватися наступний операнд у виразі, або обчислене значення може використовуватися в операторі. Наприклад, у програмі *minimdis* із підрозділу 3.3 результат виклику функції присвоюється змінної *dd*. У програмі *simpi* із прикладу 4.7 виклик функції *issimple* записана як умова в **if-операторі**, і після виклику визначається, яка з гілок повинна бути виконана. При обчисленні виразу sqr(x)+sqr(y) спочатку один за одним виконуються виклики функції sqr, потім повернуті з них значення додаються.

Конкретний вигляд точки повернення в машинній програмі ми не уточнюємо.

**2. Локальна пам'ять процесу виконання виклику підпрограми**

У підрозділі 3.4 ми вже говорили про те, що параметрам-значенням підпрограми при виконанні її виклику зіставляються власні ділянки пам'яті, тобто змінні. У блоці підпрограми можуть бути означені власні імена змінних – ділянки пам'яті зіставляються їм так само. Крім того, є ділянка пам'яті з точкою повернення. Сукупність усіх цих ділянок пам'яті називається ***локальною пам'яттю процесу виконання виклику підпрограми***. Її часто неточно називають локальною пам'яттю підпрограми. Будемо грішити цим і ми. Змінні в ній також називаються ***локальними***.

Якщо підпрограма є функцією, то до її локальної пам'яті додається змінна для збереження значення, що повертається з її виклику. Можна сказати, що ця змінна ставиться у відповідність імені функції.

Таким чином, наприклад, локальна пам'ять процедури *swap* із підрозділу 3.4 (у її остаточному вигляді) складається з єдиної змінної, зіставленої імені *t*, а пам'ять функції *dd* із програми *minimdis* (підрозділ 3.3) – із змінних, зіставлених іменам *xx1*, *yy1*, *xx2*, *yy2* і *dd*. А також указівки на точку повернення.

**3. Підстановка аргументів на місце параметрів**

Перед тим, як починається виконання операторів тіла підпрограми, для неї виділяється локальна пам'ять і в ній запам'ятовується точка повернення. Потім обчислюються значення тих аргументів у виклику, які відповідають параметрам-значенням. Ці значення присвоюються локальним змінним, які поставлено у відповідність параметрам-значенням.

*Присвоювання значень аргументів, тобто копіювання їх у локальну пам'ять*, називається ***підстановкою аргументів на місце параметрів за значенням***.

Підстановка аргументів на місце параметрів-змінних відбувається зовсім іншим шляхом. Нагадаємо, що таким аргументом може бути тільки ім'я змінної (або інше її позначення, наприклад, у вигляді поля структури). Ніякого обчислення її значення, тобто розіменування аргументу, не відбувається.

*Ця змінна, тобто ділянка пам'яті, ставиться у відповідність параметрові підпрограми*. Це називається ***підстановкою аргументу на місце параметра за посиланням***, або ***за адресою***. У результаті такої підстановки

*ім'я параметра-змінної перетворюється на вказівку на ділянку пам'яті, яка поставлена у відповідність аргументу*.

Вказівка на ділянку пам'яті називається ***посиланням на неї***, або її ***адресою***. Як саме вказівка на аргумент стає відомою у процесі виконання підпрограми, ми не уточнюємо. Для цього нам довелося б занадто глибоко забратися в устрій комп'ютера і машинної мови.

**4. Процес виконання виклику підпрограми**

Опишемо процес виконання виклику підпрограми (вона називається такою, що ***викликається***, а програма або підпрограма, що містить її виклик – такою, що ***викликає***).

1. Виділяється локальна пам'ять підпрограми, що викликається, а точніше, пам'ять процесу виконання виклику.
2. Обчислюється й запам'ятовується в локальній пам'яті підпрограми, що викликається, точка повернення в ту, що викликає.
3. Обчислюються значення аргументів, відповідних параметрам-значенням, і посилання на пам'ять аргументів, відповідних параметрам-змінним. Виконується підстановка аргументів.
4. Лише після цього виконуються оператори тіла підпрограми. Зокрема, у локальній пам'яті запам'ятовується значення, що повертається з виклику функції.
5. Значення змінної з ім'ям функції копіюється з локальної в пам'ять програми, що викликає, або підпрограми, відведену під значення, що повертається. Процес виконання виклику підпрограми закінчується і продовжується виконання програми чи підпрограми, що викликає, із точки повернення.

Як уже говорилося в підрозділі 3.4, усяка зміна значення параметра-змінної є насправді зміною відповідного аргументу. Після закінчення виклику результати обробки параметрів-змінних залишаються в аргументах, чого не можна сказати про параметри-значення.

Змінні локальної пам'яті підпрограми не зіставлені ніяким іменам підпрограми, у якій записано виклик. А це значить, що після закінчення виконання виклику *локальна пам'ять стає недоступною в підпрограмі, що викликає*, і можна вважати, що *вона* звільняється. А раз так, то її можна наново зайняти під локальні змінні при виконанні наступного виклику якої-небудь підпрограми.

І останнє зауваження про те, як при написанні підпрограми вибирати вид параметра – параметр-значення чи параметр-змінна.

*Якщо після виклику підпрограми повинно використовуватися нове значення аргументу, одержане при виконанні виклику, то відповідний параметр слід означити як параметр-змінну*. Наприклад, як параметри в остаточному варіанті процедури swap із підрозділу 3.4.

*Якщо після виклику підпрограми використовується старе значення аргументу або аргументом може бути довільний вираз, то йому має відповідати параметр-значення*. Як параметр функції *issimple* з прикладу 4.7.

З останнього правила, утім, є винятки, пов'язані з параметрами та аргументами, що є масивами. Про це йтиметься в підрозділі 12.1.

**Приклад 1.** Розглянемо імітацію виконання програми з викликами підпрограм. Як і в підрозділі 7.1, будемо позначати локальну змінну, наприклад, *L* підпрограми *F,* як *F.L*, відрізняючи її від однойменної змінної програми. Вказівку точки повернення позначимо буквами *ТП*. Процес виконання програми

**program** nested ( input, output );

**var** a, b : integer;

**function** f ( x : integer ) : integer;

**begin**

x := x + 1; f := x

**end**;

**function** g (**var** x : integer ) : integer;

**begin** x := x **div** 2; g := x **end;**

**begin** a := 12; b := f ( g ( a ) ) **end.**

відобразимо такою таблицею:

|  |  |
| --- | --- |
| **Що виконується** | **Стан пам'яті** |
|   | a b |
| a := 12 | 12 ? |
| починається b := f(g(12)) | 12 ? |
| виклик f(g(12)) | 12 ? | f. x f. f ТП  |
|   | 12 ? | ? ? b:= |
| починається f. x := g(12) | 12 ? | ? ? b:= |
| виклик g(12) | g. x ? | ? ? b:= | g. g ТП |
| g. x := 12 | 12 ? | ? ? b:= | ? f. x:= |
| g. x := g. X div 2 | 6 ? | ? ? b:= | ? f. x:= |
| g. g := g. X | 6 ? | ? ? b:= | 6 f. x:= |
| повернення з g | 6 ? | ? ? b:= | 6 f. x:= |
| закінчення f. x := g(12) | 6 ? | 6 ? b:= |
| f. x := f. x + 1 | 6 ? | 7 ? b:= |
| f. f := f. X | 6 ? | 7 7 b:= |
| повернення з f | 6 ? | 7 7 b:= |
| закінчення b := f(g(12)) | 6 7 |

Як видно з таблиці, перед виконанням виклику f(g(a)) у локальній пам'яті функції *f* запам'ятовується точка повернення – присвоювання b:=f(g(a)). Виконання виклику f(g(a)) починається обчисленням значення аргументу, відповідного її параметру *x* – виразу g(a). При цьому запам'ятовується точка повернення – присвоювання f.x:=g(a) – і починається виконання виклику функції *g* з аргументом *a*. Цей аргумент підставляється за посиланням, і зміна *g.x* є зміною *a*. З виклику функції *g* повертається значення 6 і присвоюється параметру *f.x*. Пам'ять функції *g* звільняється. Тільки тепер починається виконання операторів функції *f*. У результаті в програму повертається значення 7 і присвоюється змінній *b*. Пам'ять функції *f* звільняється.

**Задачі**

**1.**\* Чи є правильною програма з прикладу 8.1, якщо замість оператора b:=f(g(a)) у її тілі записано

а) b:=g(f(a)); б) b:=f(a+33); a:=g(b); в) b:=f(g(a)+1); b:=f(f(a)+1).

Імітувати виконання нової програми, якщо вона правильна.

**5. Автоматична пам'ять та програмний стек**

Нагадаємо, що виконання програми починається після її завантаження в пам'ять комп'ютера. При цьому відбувається виділення пам'яті під її змінні. Ці змінні доступні з програми протягом усього часу її виконання, і тому називаються ***статичними***. Область пам'яті, що виділяється під програму та її змінні, також називається ***статичною***. Локальним іменам і параметрам-значенням підпрограм, як правило, зіставляють змінні з іншої області пам'яті – ***автоматичної***. Ця назва пояснюється тим, що при виконанні викликів підпрограм ця пам'ять виділяється та звільняється "автоматично", без явних вказівок у програмі.

Як ми вже говорили в підрозділі 7.1, локальні імена, означені в підпрограмах, можуть збігатися з іменами в програмі та інших підпрограмах, оскільки *вони позначають цілком різні об'єкти в різних областях пам'яті*. Тому на локальні імена в підпрограмах немає ніяких обмежень. Зрозуміло, те саме ім'я в межах тіла підпрограми не повинно позначати різні об'єкти, наприклад, локальну змінну та іншу підпрограму. Не можна також означати ім'я самої підпрограми в її ж блоці.

Звернімо увагу на таблицю, що відбиває імітацію програми з прикладу 8.1. Спочатку виділяється локальна пам'ять функції *f*, потім – функції *g*, але звільнення відбувається в зворотнім порядку.

Виділення та звільнення ділянок пам'яті при виконанні викликів підпрограм відбувається за принципом "останнім зайнятий – першим звільнений". Подібно тому, як заштовхують патрони в магазин автомата й вистрілюють. Останній із них вистрілюється першим, а перший, що на самому дні магазина, – останнім. Можна вистрілити один патрон із магазина і додати згори новий. Цьому відповідає закінчення виклику підпрограми, записаного в тілі деякої підпрограми, і початок виконання наступного виклику з цього ж тіла.

Якщо складати аркуші паперу в стопку і брати їх тільки згори, то лист, що потрапив у стопку останнім, забирається першим. По-англійському така стопка називається stack (стек), а кладуться і забираються аркуші за принципом "**Last** **In** – **First** **Out**" (LIFO), тобто "*першим у – останнім із*". Тому автоматична пам'ять програми називається ***програмним стеком***. Таблиця в прикладі 8.1 показує, як зростає й зменшується програмний стек при виконанні викликів підпрограм.

**6. Зберігання локальних змінних між викликами підпрограми**

У деяких задачах необхідно використовувати змінні, які означені й доступні в підпрограмі, але зберігають своє значення від її попереднього виклику до наступного.

*Ці змінні не можна розміщати в автоматичній пам'яті*, оскільки вона недоступна після виконання виклику підпрограми. *Ці змінні не можна означати в програмі*, оскільки тоді вони доступні не тільки в потрібній підпрограмі. Де і як їх означати?

Виходом є означення цих змінних *у підпрограмі як статичних*. Саме такі змінні розміщаються в *статичній пам'яті програми* разом із змінними програми, *але доступні тільки під час виконання виклику* тієї підпрограми, у якій означені. Вони називаються ***локальними*** ***статичними змінними***. Розглянемо задачу, у якій використовуються такі змінні.

**Приклад 2.** У багатьох задачах, від моделювання природних або соціальних процесів і до розкладання карт, потрібні послідовності чисел, що належать деякій множині, але більше ніяк не пов'язані одне з одним. Такі числа називаються ***випадковими***.

Поява таких послідовностей імітується за допомогою підпрограми, що називається ***генератором псевдовипадкових чисел***. Кожне нове число утворюється застосуванням рекурентного співвідношення до попереднього. Очевидно, що така послідовність не випадкова, але вона "виглядає, як випадкова". Лише перше число можна вважати випадковим, тому що воно надходить від одного з зовнішніх пристроїв комп'ютера. У реальних генераторах це, як правило, таймер (машинні часи), а тут це буде клавіатура.

У багатьох генераторах використовується множина цілих чисел від 0 до *m*-1, де *m* – достатньо велике число. Нове число визначається за попереднім застосуванням рекурентного співвідношення вигляду

*Vi*+1=( *a*\**Vi* + *c* ) **mod** *m*.

Цей дуже простій у реалізації метод породження наступного числа називається ***лінійним*** ***конгруентним***. Ми не будемо тут досліджувати, при яких значеннях параметрів *a*, *c* і *m* і чому утворюється послідовність, схожа на випадкову. Про це написано дуже багато, і достатньо заглянути в книгу Д.Кнута [Кнут, т.2]. Для прикладу візьмемо *m*=1001, *a*=21, *c*=57.

Наша задача – написати функцію, що задає обчислення та повернення наступного елемента послідовності за умови, що відомий попередній. Дамо їй ім'я *NextRand* – скорочення від "Next Random", або "наступне випадкове".

Очевидною є функція, з якої повертається значення її параметра-змінної, а він змінюється так, як задано рекурентним співвідношенням. Проте в програмі з такою функцією необхідно означати змінну, що була б аргументом у викликах функції. Але ця змінна доступна для зміни в самій програмі й іншим її підпрограмам, і таке рішення не є задовільним.

Ту змінну, чиїми значеннями є послідовні псевдовипадкові числа, означимо в нашій функції як *статичну*. Те, що змінна статична, якимось чином треба указати в її означенні. Наприклад, у мові Сі та деяких діалектах мови Паскаль там записуються додаткові позначення, щось на зразок

**static** **var** V : integer.

У діалекті Турбо Паскаль локальну статичну змінну задає *означення* з *ініціалізацією* (п.2.3.6), записане в підпрограмі. Нагадаємо його дещо дивний вигляд:

**const** *ім'я* : *тип* = *ініціалізуюче-значення*;

З ним наша функція виглядає так:

**function** NextRand : integer;

**const** V : integer = 0; **const** m=1001; a=21; c=57;

**begin**

V := ( a\*V+ c ) **mod** m; NextRand := V

**end**;

Як видно, першим псевдовипадковим числом завжди буде 0. Але в задачі було потрібно, щоб перше число надходило "із зовнішнього світу". Змінимо функцію так, що при виконанні її першого виклику відбувається запит на введення першого числа з клавіатури, а при наступних – обчислення за рекурентним співвідношенням:

**function** NextRand : integer;

**const** V : integer = 0; First : Boolean = **true**;

**const** m=1001; a=21; c=57;

**function** Rand1: integer;

**var** t : integer;

**begin**

write('Задайте ціле від 0 до ', m-1, '>'); readln(t);

Rand1 := t

**end**;

**begin**

**if** First **then**

**begin** First := **false**; V := Rand1 **end**

**else** V := ( a\*V+ c ) **mod** m;

NextRand := V

**end**;

Тепер можна записувати цю функцію та її виклики в програмах, де потрібно імітувати появу випадкових чисел. Проте краще помістити цю функцію в модуль (п.7.2 ) з ім'ям, наприклад, *Randoms*, транслювати його та у програмі вказувати лише його використання: **uses** Randoms.

Використання модулів дозволяє розв'язати нашу задачу взагалі без використання локальних статичних змінних. Справа в тім, що *змінні, означені в модулі*, як і змінні програми, *є статичними*. Тому модуль можна записати в такому вигляді:

**Unit** Randoms;

**Interface**

**function** NextRand : integer;

**Implementation**

**const** m=1001; a=21; c=57; **var** V : integer; First : Boolean;

**function** Rand1: integer;

**var** t : integer;

**begin**

write('Задайте ціле від 0 до ', m-1, '>'); readln(t);

Rand1 := t

**end**;

**function** NextRand;

**begin**

**if** First **then**

**begin** First := **false**; V := Rand1 **end**

**else** V := ( a\*V+ c ) **mod** m;

NextRand := V

**end**;

**Begin** First := **true**

**End**.

Як бачимо, змінні *V* та *First* стали глобальними в модулі і доступними в його підпрограмах, але за його межами їх "не видно". Можна сказати, що їх означення *локалізовані в модулі*.

**Задачі**

**2.** Переробити модуль Randoms так, щоб він задавав породження псевдовипадкової послідовності

а) дійсних чисел з проміжку [0; 1]; б) дійсних чисел з проміжку [*c*; *d*].

**3.** Переробити модуль Randoms так, щоб зовні його можна було використовувати функцію Rand1. Написати програму обчислення ***середнього арифметичного значення M*** та дисперсії *D* псевдовипадкової послідовності дійсних чисел {*Xi*} з проміжку [0;1]. ***Дисперсія*** – це середнє арифметичне квадратів різниць між числами та *M*:

*D* = ( ( *X*1 - *M* )2 + … + ( X*N* - *M* )2 )/*N*,

де *N* – загальна кількість елементів. Добрий генератор має давати значення *M* і *D*, близькі до 1/2 та 1/12 відповідно. Але не всякий генератор, що дає такі середнє значення та дисперсію, є добрим.

**4.** Площу *S* двовимірної геометричної фігури обмежених розмірів можна обчислити в такий спосіб. Розташувати фігуру всередині іншої, площа *S*2 якої обчислюється досить просто за формулами. Нею, як правило, є прямокутник у декартовій системі координат. Вибрати довільно *N* точок із другої фігури, указуючи їхні координати, та обчислити кількість *K* тих із них, що належать першій фігурі. Відношення *K/N* буде наближенням до *S/S*2. Координати точок обчислюються за допомогою випадкових чисел, тому цей спосіб називається ***методом Монте-Карло* –** за назвою курорту, де грали в рулетку.

Написати програму обчислення за методом Монте-Карло наближення до площі кола, заданого координатами центру та радіусом.

**5.** Реалізувати генератор псевдовипадкових чисел, оснований на рекурентному співвідношенні *Vi*=(*a*\**Vi*-1+*b*\**Vi*-2+*c*) **mod** *m*. Перші два значення *V*1 і *V*2задаються випадково. Параметри підібрати самостійно. Використати генератор в задачах 8.3–8.4.

**7. Підпрограми як параметри**

*У мові Паскаль параметрами підпрограм можуть бути не тільки змінні, але й підпрограми*.

Розглянемо приклад. Нам потрібно надрукувати три таблиці значень трьох математичних функцій на заданому відрізку [*a*; *b*], де *a*>0, у точках, розташованих із заданим кроком *h*. Функції такі:

sh(*x*) = (*ex-e-x*)/2, ch(*x*) = (*ex*+*e-x*)/2,

th(*x*) = (*ex-e-x*)/(*ex*+*e-x*) = (*e*2*x*-1)/(*e*2*x*+1) .

У програмі можна записати функції з іменами *sh*, *ch* і *th*, що задають необхідні обчислення, та означити змінні *a*, *b*, *h*, *n*, *k*, *x*. Тоді оператори тіла програми можуть мати такий вигляд:

Readln(a, b, h);

n:= trunc((b-a)/h);

**for** k:=0 **to** n **do**

**begin** x:=a+k\*h; writeln(x, ' ', sh(x)) **end**;

**for** k:=0 **to** n **do**

**begin** x:=a+k\*h; writeln(x, ' ', ch(x)) **end**;

**for** k:=0 **to** n **do**

**begin** x:=a+k\*h; writeln(x, ' ', th(x)) **end**;

Достатньо нудно. Якби в нас була підпрограма (процедура) з ім'ям tabf, параметризована не тільки відрізком і відстанню між точками, але також і функцією, то замість трьох циклів достатньо було б написати лише

tabf(sh, a, b, h); tabf(ch, a, b, h); tabf(th, a, b, h);

Займемося написанням процедури *tabf*. Нехай *f* буде ім'ям параметра, що позначає функцію, яка табулюється; імена й зміст інших параметрів очевидні. Для початку маємо

**procedure** tabf( *означення параметра f*; a, b, h : real);

**var** k, n : integer; x : real;

**begin** n := trunc((b-a)/h);

**for** k:=0 **to** n **do**

**begin** x:=a+k\*h; writeln(x, ' ', f(x)) **end**;

**end**;

Уточнимо тепер *означення параметра f*. *У стандарті мови Паскаль* воно мало б вигляд заголовка функції, ім'я параметра в якій не має значення, наприклад,

**function** f(xxxx : real) : real.

Зауважимо, що функції *sh*, *ch* і *th* слід було б означити з аналогічними заголовками, тобто, наприклад,

**function** sh(x : real) : real.

Отже, заголовком процедури було б

**procedure** tabf(**function** f(xxxx : real) : real; a, b, h : real);

Якби *f* була не функцією, а процедурою, ми записали б відповідно

**procedure** tabf( **procedure** f(…);…)…

У стандарті мови Паскаль *параметри-підпрограми можуть мати параметрами тільки параметри-значення, а параметри-змінні та підпрограми не допускаються*. Втім, параметр-підпрограма може й зовсім не мати параметрів.

*У мові Турбо Паскаль типи параметрів у заголовках підпрограм можна задавати тільки іменами*. Інші вирази заборонені. Тому тут потрібно заздалегідь означити ***тип підпрограми*** і використовувати його в заголовку *tabf*:

**type** typfun = **function** (xxxx : real) : real; {ім'я функції відсутнє ! }

**procedure** tabf( f : typfun; a, b, h : real);

Однією з особливостей роботи з системою Турбо Паскаль є використання так званих ***директив транслятора***. Вони записуються в коментарі з символом **$** на початку і задають спеціальні режими роботи транслятора. Не поринаючи у подробиці, скажемо лише, що якщо підпрограма використовується як аргумент у викликах інших підпрограм, то перед нею треба написати директиву {$F+}, а після – {$F-}. У нашому прикладі означення імен функцій sh, ch і th слід узяти в "директивні дужки":

{$F+}

**function** sh(x : real ) : real; … **end**;

**function** ch(x : real ) : real; … **end**;

**function** th(x : real ) : real; … **end**;

{$F-}

У мові Турбо Паскаль підпрограми, що використовуються як аргументи, можуть мати параметри всіх трьох видів, а не лише параметри-значення. Можливо також використання *змінних типу підпрограма*. Наприклад, ми могли б у нашому прикладі означити "змінну типу функція"

**var** zz : typfun;

і використовувати її далі в програмі:

**for** m:=1 **to** 3 **do**

**begin if** m=1 **then** zz:=sh **else**

**if** m=2 **then** zz:=ch **else** zz:=th;

tabf(zz, a, b, h)

**end**;

У діалекті Турбо Паскаль не можна використовувати стандартні математичні функції як аргументи у викликах підпрограм. Обійти це обмеження нескладно. Достатньо написати й використати власну функцію з іншим ім'ям, наприклад,

**function** sinmy(x : real) : real;

**begin** sinmy := sin(x) **end**;

**Задачі**

**6.** Написати процедуру, що реалізує метод половинного ділення (задача 5.4) і має один із параметрів типу **function** ff(x : real) : real.

**7.** Написати програму обчислення кореня рівняння sinx- *a* = 0 на відрізку [- /2;  /2], де -1<*a*<1, із використанням підпрограми з попередньої задачі.