5. Сложные операторы

Глупцы игнорируют сложность. Прагматики терпят её. Некоторые могут избегать её. Гении её устраняют.

Алан Перлис,
первый лауреат премии Тьюринга

Как уже говорилось, сложные операторы, в отличие от простых, содержат внутри себя хотя бы один оператор. Всего в стандарте Паскаля четыре сложных оператора, сейчас мы рассмотрим их синтаксис, семантику и прагматику.

5.1. Составной оператор

Программы предназначены для чтения людьми и только случайно для выполнения компьютерами.

Дональд Эрвин Кнут

Сначала синтаксис:

<составной оператор> ::= **begin** <оператор>{;<оператор>}… **end**

Отметим, что синтаксис составного оператора (compound statement) полностью совпадает с синтаксисом раздела операторов, да и семантика у него такая же. Когда в программе нет операторов перехода, то выполнения составного оператора заключается в выполнении в порядке их следования и *по одному разу* всех входящих в него операторов. Составной оператор можно изобразить так (см. рис. 5.1)

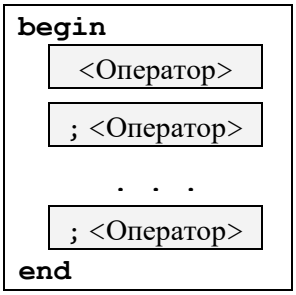


Рис. 5.1. Структура составного оператора.

Прагматика составного оператора очень проста. Во многих местах по синтаксису Паскаля может стоять ровно один оператор. Когда по смыслу алгоритма в этом месте надо выполнить несколько действий, то надо поставить один оператор, но составной, внутрь которого и включить все эти действия. Как видим, составной оператор отражает свойство *структурности* алгоритма. Служебные слова **begin** и **end** часто называют операторными скобками. Как и выражение любой сложности в круглых скобках сразу становится простым, так и произвольное число операторов в операторных скобках сразу становится одним оператором.

Понятие составного оператора впервые появилось в языке Алгол-60, там он был такой же, как и в Паскале. А вообще в разных языках составные операторы оформляются по разному. Например, в языке C вместо операторных скобок **begin** и **end** используются фигурные скобки **{ }** (как комментарий в Паскале). Надо, однако, учесть, что составной оператор в C является блоком, кроме того, каждый оператор в этом языке заканчивается точкой с запятой. А вот в языке Python операторных скобок вообще нет, точнее, все операторы, входящие внутрь составного, задаются *отступом* он начала строки на некоторое, *строго одинаковое* число пробелов.

5.2. Оператор выбора

Когда необходимо сделать выбор, а вы его не делаете, – это тоже выбор.

Уильям Джеймс
«Принципы психологии»

Абсолютной свободы не существует: есть лишь свобода выбора, а, сделав выбор, ты становишься заложником своего решения.

Пауло Коэльо. «Заир»

Внутрь каждого оператора выбора (selection statement) входит не менее одного оператора, и условие, как выбрать для выполнения ровно один из них. В Паскале (и в большинстве других языков программирования) есть два вида таких операторов.

1. Условный оператор. Внутри у него два оператора, они называются ветвями (branches), и надо выбрать для исполнения один из них. Синтаксис:

```
<условный оператор> ::= if  <логическое выражение>
                          then  <оператор>
                        [ else  <оператор>  ]
```

Семантика этого оператора простая. Сначала вычисляется логическое выражение, если оно истинно, то выполняется оператор, стоящий после **then**, иначе после **else**. В том случае, когда после **else** стоит *пустой* оператор, то слово **else** можно опустить., это называется *сокращённой формой* условного оператора:

```
if  <логическое выражение>  then  <оператор>
```

Прагматика этого оператора очевидна. Условный оператор можно изобразить так (см. рис. 5.2)

Рис. 5.2. Структура условного оператора.

В Паскале операторы не являются выражениями, т.е. не имеют значений, их нельзя куда-нибудь вычислить, присвоить, распечатать и т.д. В некоторых языках, кроме условных операторов, существуют и *условные выражения*, например, в языке Алгол-60:

```
x:=if x<0 then -x else 2*x
```

В языках C и PascalABC.NET существует похожая, так называемая тернарная операция:

```
x = x<0 ? -x : 2*x  // язык C
x:=x<0 ? -x : 2*x  // PascalABC.NET – близнец C
```

Всё значительно усложняется, если разрешить метки и оператор **goto**. В этом случае метку перехода можно поставить на оператор в ветви **then** или в ветви **else** и перейти на эти операторы, не вычисляя логического выражения, например:

```
goto L; if x<3 then L: x:=5;
if x<3 then goto L else L: x:=5;
```

Языки, которые используют **goto**, либо прямо запрещают это, либо явно говорят, что результат выполнения такой программы не определён. К сожалению, в языке Free Pascal это можно делать, если разрешить использование оператора **goto** директивой {$**goto**+}.

2. Оператор **case**. Этот оператор может содержать произвольное число операторов-ветвей, он используется, в основном для обработки символьных данных. Синтаксис:

```
<оператор case> ::=
    case <выражение> of
         <ветвь>;{<ветвь>;}...
    end
<ветвь>  ::= <выбор>:<оператор>
          {;<выбор>:<оператор>}...
<выбор>  ::= {<константа>[..<константа>]}
            {,<константа>[..<константа>}...
```

Оператор **case** можно изобразить, как показано на рис. 5.3.

Здесь выражение и все *константы* должны буть одного и того же скалярного типа (кроме вещественного ⚠), если указаны две константы, то они должны задавать непустой диапазон значений данного типа. При совпадении констант в двух и более ветвях фиксируется ситаксическая ошибка.

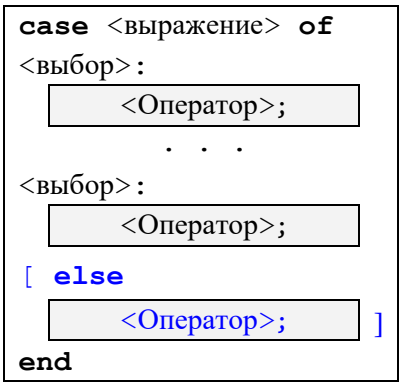


Рис. 5.3. Структура оператора **case**.

Выполнение этого оператора начинается с вычисления выражения, затем значение вычисленного выражения последовательно проверяется на совпадение с константой ветви (или на попадание в диапазон констант ветви). При успешном сравнении выполняется (один) оператор в выбранной ветви, а если все сравнения выражения с константами в ветвях были безуспешними, то действие оператора в стандарте Паскаля не определено.

В языке Free Pascal в этот оператор можно добавить в конце дополнительную ветвь

else <оператор>

и тогда при безуспешном сравнении значения выражения с константами в ветвях выполнится оператор в ветви **else**, а если этой ветви нет, то этот оператор **case** «ничего не делает». Например:

```
  var x: char;
begin read(x);
  case pred(succ(x)) of {выражение 🙂}
   '0'..'9':
      Write('Цифра');
   'a'..'z','A'..'Z':
      Write('Латинская буква');
   '.',',',':','!','?','-':
      Write('Знак препинания');
   '@':
      Write('"Собака" или "Лягушка"');
   else
      Write('Неизвестный символ!')
  end {case}
end.
```

5.3. Оператор цикла

Иногда мне кажется, что единственным универсумом в программировании является цикл.

Алан Джей Перлис,
первый лауреат премии Тьюринга

1. Начальник всегда прав.
2. Если начальник не прав, см. Пункт 1.
Цикл с постусловием 🙂

Операторы цикла (iterative statements) содержат внутри себя один или несколько операторов, которые могут выполняться более одного раза. В большинстве языков программирования существуют три вида операторов цикла. Здесь надо учесть, что у них разная прагматика и в каждом конкретном

случае обычно один из циклов более эффективен, чем два других. Оператор (или операторы), стоящие внутри цикла, называются телом цикла.

1. Цикл с предусловием. Синтаксис:

```
<цикл с предусловием> ::=
      while  <логическое выражение>  do  <оператор>
```

Здесь тело цикла является *одним* оператором. Выполнение этого цикла начинается с вычисления логического выражения. Когда это выражение истинно, то выполняется оператор в теле цикла, после чего опять вычисляется логическое выражение. Когда логическое выражение принимает ложное значение, то весь оператор цикла считается выполненным. Этот оператор цикла можно изобразить так (см. рис. 5.4).

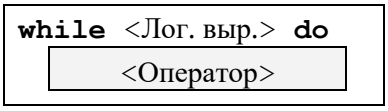


Рис. 5.4. Структура цикла с предусловием.

Из семантики этого цикла следует и его прагматика. Когда по условию задачи тело цикла должно быть выполнено неизвестное заранее число раз, причём возможен и случай, когда тело цикла не надо выполнять ни разу, то следует использовать цикл с предусловием. Поймите также, что число повторений тела цикла в момент начала выполнения этого оператора может быть ещё неизвестно.

2. Цикл с постусловием. Синтаксис:

```
<цикл с постусловием> ::=
      repeat  <оператор>{;<оператор>}...
      until  <логическое выражение>
```

Здесь в теле цикла стоит один или несколько операторов, причём служебные слова **repeat** и **until** выполняют роль операторных скобок (аналогичных **begin** и **end**). Выполнение этого цикла начинается с выполнения по одному разу всех операторов из тела этого цикла (напоминаем, что оператора **goto** нет). Затем вычисляется логическое выражение. Когда это выражение ложно, то снова выполняется тело цикла и т.д. Оператор считается выполненным, когда очередное вычисление логического выражение дало значение истина (**true**). Этот оператор цикла можно изобразить так (см. рис. 5.5).

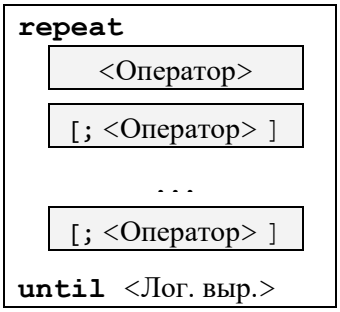


Рис. 5.5. Структура цикла с постусловием.

Из семантики этого цикла следует и его прагматика. Когда по условию задачи тело цикла должно быть выполнено неизвестное заранее число раз, однако не менее одного раза, то следует использовать цикл с постусловием.

Заметим, что слова **while** и **until** переводятся на русский язык одним словом «пока», но на английском у них разная семантика «пока да» и «пока нет». Проще всего запомнить правило, что для логического значения «истина» исполнитель алгоритма всегда идём вниз, но для цикла с предусловием это продолжение цикла, а для постусловие – окончание.

Отметим, что аналогичный цикл с постусловием языка С записывается как:

do {<операторы>} **while** <логическое выражение>

и выходит из цикла по логическому значению **false**, т.е. как и цикл **while** с *предусловием*, цикл **while** с *постусловием* повторяется, пока значение логического выражения истинно.

Цикл с постусловием более эффективно реализуется на языке машины, чем цикл с предусловием. Исходя из этого, все компиляторы с языков высокого уровня предпочитают преобразовывать в похожий вид даже циклы с предусловием.

3. Цикл с параметром. У этого оператора сложный синтаксис и семантика. Синтаксис:

```
<цикл с параметром> ::=
    for <параметр цикла>:=<выражение1>{to|downto} <выражение2>
        do <оператор>
<параметр цикла> ::= <простая дискретная переменная>
```

Для компактности обозначим параметр цикла как `i`, выражения как `V1` и `V2`, а оператор как `S`, тогда цикл перепишется в таком виде:

```
for i:=V1 {to|downto} V2 do S
```

Итак, параметром цикла (loop variable) является простая дискретная переменная. Слово «простая переменная» означает, что она не является составной частью более сложной переменной (например, элементом массива или полем записи). Дискретные типы нам уже известны, у каждой величины такого типа (кроме первой и последней в своём типе) обязательно есть следующее и предыдущее значения, их можно вычислить с помощью стандартных функций `succ` и `pred`. В Паскале дискретными являются только целые, символьные, логические и перечислимые типы. Выражения `V1` и `V2` должны быть совместимы по присваиванию с переменной `i` (самое время вспомнить, что это такое).[1] Тело цикла является одним оператором.[i] [см. сноску в конце главы]

Сначала рассмотрим случай, когда используется служебное слово **to**, тогда цикл примет вид

```
for i:=V1 to V2 do S
```

Для объяснения работы этого цикла напишем эквивалентный фрагмент программы на Паскале (т.е. наш цикл будет выполняться точно так же, как этот фрагмент):

```
create(temp); { создать переменную temp }
❶ temp:=V2; i:=V1;
repeat
  if ❷ i<= temp then S;
  if ❸ i<temp then i:=succ(i)
until ❹ i>=temp;
delete(temp); { уничтожить переменную temp }
i:=Random; { i:=случайное значение своего типа }
```

Выполнение цикла с параметром начинается с порождения служебной переменной `temp` (ей отводится память), она такого же типа, как и переменная `i`. Зачем это делается? В стандарте Паскаля сказано, что при выполнении цикла с параметром значение выражения `V2` вычисляется только один раз. Как вычислить выражение один раз, а сравнивать его значение с параметром `i` более одного раза (у нас это точки ❶, ❷ и ❸)? Это можно сделать, только вычислив значение и запомнив его в какой-то переменной, что мы и сделали в точке ❶. После цикла эта переменная должна быть уничтожена, т.е. занимаемая ею память освобождена. Ясно, что сам цикл может быть телом другого цикла, и, если переменную порождать в цикле и не уничтожать, то память вскоре исчерпается.

Далее, в стандарте Паскаля сказано, что значения параметра цикла `i` после выхода из этого цикла не определено, т.е. имеет случайное значение.[2] И, наконец, стандарт Паскаля требует, чтобы параметр цикла явно не изменялся в теле этого цикла, т.е. нельзя, например, писать

```
for i:=1 to 10 do i:=i+1 { ОШИБКА! }
```

Компилятор языка Free Pascal предпринимает «титанические» усилия, чтобы не допустить изменения параметра цикла в теле этого оператора (зафиксировав синтаксическую ошибку). Нельзя из

---

[1] В языке Free Pascal параметр цикла может быть как локальной, так и глобальной переменной, но не может быть так называемой типизированной константой (см. разд. 8.1). Смысл этого ограничения не ясен.

[2] При разработке Паскаля были соображения, почему цикл с такой семантикой будет эффективным при реализации на ЭВМ, мы это рассматривать не будем. В языке Free Pascal параметр цикла просто сохраняет последнее присвоенное ему значение.

тела цикла передать параметр в подпрограмму по ссылке, изменить переменную, наложенную в памяти на параметр цикла (`absolute`) и т.д. Но можно, например, изменить параметр цикла так:

```
var j:↑integer; { ↑ будем изучать в главе 12 }
j:=Addr(i);
for i:=1 to 5 do j↑:=j↑-1; { зациклились 🙄 }
```

Из схемы выполнения цикла **for** видно, что тело этого цикла может не выполняться ни одного раза, т.е. по существу это цикл с предусловием. Далее, можно заметить очень важное свойство цикла с параметром: придя на начало такого цикла, программа абсолютно точно знает, сколько раз надо выполнить тело цикла (iteration count). Как говорят, это цикл с *известным числом повторений* (definite loop). Например, тело цикла

```
i:=2; for i:=1 to i+1 do S
```

будет выполнено три раз, а тело цикла

```
i:=2; for i:=0 to i-3 do S
```

не выполнится ни одного раза. Обязательно поймите эти примеры.

И, наконец, параметр цикла должен быть локальным, т.е. описанным в том же блоке, где находится и оператор цикла. К вопросу локальности мы вернёмся при описании подпрограмм.

Циклы с параметром играют важную роль в теории алгоритмов. Программы, которые не содержат других видов циклов (а также рекурсии и операторов перехода **goto**), обладают замечательным свойством: они всегда останавливаются, т.е. не могут зациклиться ⚠. Так как это второй после деления но ноль кошмар для программистов, то видно, как просто его избежать. Циклы этого вида настолько важны, что в машинных языках существуют специальные команды цикла, позволяющие их эффективно реализовать. К сожалению, только этими циклами все задачи решить невозможно, надо обязательно использовать циклы с неизвестным заранее числом повторений (или мерзкие **goto** 🙁), либо особые, *рекурсивные* подпрограммы, их мы будем изучать позже.

Конструкция `V1 to V2` называется списком цикла, она определяет множество значений для параметра цикла. В Паскале цикл всегда идёт с шагом один, увеличиваясь посредством `succ(i)` для цикла со служебным словом **to**, и уменьшаясь посредством `pred(i)` для цикла **downto**.

В некоторых других языках можно задать произвольный шаг такого цикла, например, с шагом 3:

```
for (i=0;i<n;i+=3) < оператор >; // Язык C
do i=1,n,3                       !  Язык Фортран-90
   <операторы>
end do
```

Некоторые языки (например, модный Python и наш Free Pascal) позволяют задать список цикла в виде так называемого итератора (итерируемого множества, см. сноску [i] в конце главы), например:

```
const s='Привет, мир!';
type Week=(Mon,Tue,Wed,Thu,Fri,Sat,Sun);
     Vec=array[1..5] of integer;
     Mat=array[1..4] of Vec;
var x: array[1..100] of integer;
    c: char; i: integer; d: Week;
    a: Vec; b: Mat;
begin
  for d in Week do write('День=',d:4);
  for c in s do write('Буква=',c);
  for c in char do write('№ буквы=',ord(c));
  for i in x do write(i); { i=x[i] }
  for i in [1,3,10..50] do write(i);
  for a in b do for i in a do write(i) { i=b[j,i] };
  for i in integer do write(i); { о-о-чень длинный цикл 🙂 }
```

Итак, мы рассмотрели три основные оператора языков программирования: составной (последовательное выполнение), выбора и цикла. Ещё в 1966 году Коррадо Бём и Дэвид Харел доказали теорему

о структурированной программе (Structured program theorem), которая утверждает, что любая программа может быть написана с использованием только этих трёх операторов (без **goto** ⚠). Так появилось структурное программирование.

5.4. Структурное программирование

Сложность программы растёт до тех пор, пока не превысит способности программиста.

6-й закон машинного программирования

В начале 70-х годов прошлого века в программировании сформировалась, как говорят, *парадигма* [1] (или *методология*) структурного программирования, в соответствие с которой вся программа и её подпрограммы строятся как чисто иерархическая структура. В качестве компонент этой структуры достаточно взять всего три сложных оператора: составной (последовательное выполнение, sequence), выбора (selecnion) и цикла (iteration).[2] Главным принципом является требование, чтобы как у всего алгоритма, так и у каждой его структурной части был один вход и один выход [21].

Итак, в 1966 году была доказана теорема о структурном программировании. Для иллюстрации этими итальянскими математиками был разработан язык программирования высокого уровня с именем P, это был первый полный по Тьюрингу язык без оператора **goto**. А в 1968 году Эдсгер В. Дейкстра в статье «О вреде оператора goto» («GoTo Statement Considered Harmful») полностью обосновал эту парадигму, в частности, он показал, что качество программы обратно пропорционально наличию в ней операторов **goto** ☹.

Многие языки программирования (Java, Ruby, Python, Swift и другие) совсем не имеют оператора **goto**, в других его использование сильно ограничено (например, нельзя «выпрыгивать» из блока, передавать управление внутрь условных операторов и циклов и т.д.). Более терпимым является отношение к «замаскиро́ванным» **goto**, это такие «безметочные» операторы перехода, как **break**, **exit** и **continue**, мы ими пользоваться не будем.

Структурное программирование призвано бороться со всё возрастающей сложностью программ, оно позволяет легко понимать и модифицировать алгоритмы. Главной целью является повышение производительности труда программистов в несколько раз.

К сожалению, произвольная программа (с операторами **goto**) не может быть преобразована в структурную без дубрирования своих отдельных частей и введения новых переменных (достаточно булевских).

5.4.1. Диаграммы Насси-Шнейдермана

Чтобы поверить в алгоритм, его нужно увидеть.

Дональд Кнут

Придумано несколько способов для графического изображения структурных программ (и их фрагментов). Первоначально широко использовались так называемые блок-схемы, для них даже был разработаны стандарты ГОСТ 19.701-90 [3] и ISO 5807-85. Блок-схемы, однако, позволяют записывать и неструктурные алгоритмы (с оператором перехода), поэтому в дальнейшем были разработаны другие графические представления именно структурных алгоритмов. Особенно компактно (без стрелок!) позволяют записывать структурные алгоритмы так называемые диаграммы (схемы) Насси-Шнейдермана, иногда называемыми *структурограммами*.

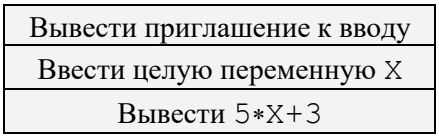
---

[1] По гречески παράδειγμα означает образец или модель, на основании которых можно строить постановку задач и методов их решений. Это нечно, признаваемое большинством учёных в качестве достоверной истины, образца для проведения исследований и решения задач.

[2] Это так называемый принцип Бома и Джакопини, правда, они вместо обобщённого оператора выбора использовали простой двоичный выбор в виде условного оператора.

[3] ГОСТ19.701-90. ЕСПД. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения.

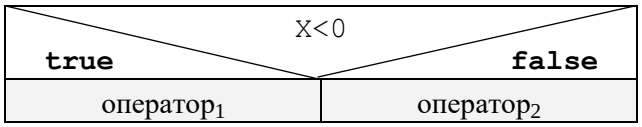
Составной оператор изображается с их помощью в прямоугольнике, разбитом на несколько частей горизонтальными линиями, в каждой части записывается один шаг (оператор):



Условный оператор, например, с таким булевским выражением

if X<0 **then** <оператор$_1$> **else** <оператор$_2$>

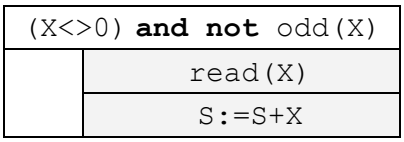
будет изображаться в виде такого прямоугольника



Цикл с предусловием

```
while (X<>0) and not odd(X) do begin
      read(X); S:=S+X
end
```

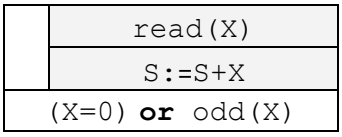
Будет изображаться так



Цикл с постусловием

```
repeat
   read(X);
   S:=S+X
until (X=0) or odd(X)
```

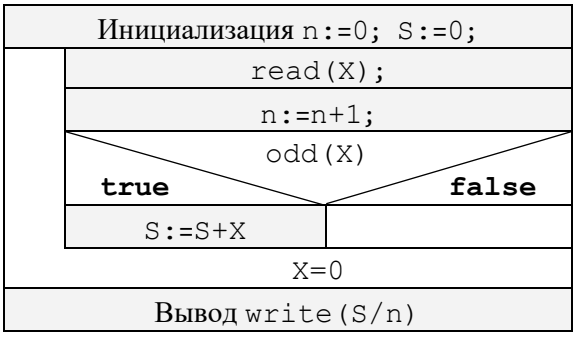
Будет изображаться так



А вот маленькая программа:

```
var n,S: integer;
begin
  n:=0; S:=0;
  repeat
    read(X); n:=n+1;
    if odd(X) then
       S:=S+X
  until X=0;
  write(S/n)
end.
```

Для неё получится такая диаграмма Насси-Шнейдермана:



Это может нравиться или не нравиться, но знать, что это такое «настоящий» программист обязан. Здесь же можно упомянуть и так называемые графические (или визуальные) языки, например, это так называемый унифицированный язык моделирования UML (Unified Modeling Language) и отечественный визуальный алгоритмический язык программирования Дракон (Интернет Вам в помощь ☺).

5.5. Оператор присоединения

Жизнь такова, какова она есть и больше никакова.

Владимир Голованов
из ранних стихов

Последний сложный оператор Паскаля называется оператором присоединения, он отсутствует в большинстве языков программирования. Мы изучим этот оператор позже в главе 9, вместе со сложным типом данных под названием запись.

Вопросы и упражнения

Не бойтесь поднапрячь мозги! Они от этого не лопнут.

Курт Воннегут.
«Колыбель для кошки»

1. Для чего нужен пустой оператор?
2. Чем сложные операторы отличаются от простых?
3. Для чего нужен составной оператор?
4. Что такое ветвь условного оператора?
5. Почему в языке только один оператор присваивания, но три оператора цикла?
6. Когда в программе надо употреблять цикл с предусловием, а когда с постусловием?
7. Когда в программе надо употреблять цикл с параметром?
8. Объясните, почему, когда в программе только циклы с параметром (и нет операторов перехода и рекурсивных подпрограмм), то она не может зациклиться.
9. Что такое список цикла?
10. В чём заключается смысл структурного программирования?

---

[i] Для продвинутых читателей. Параметр цикла является частным случаем так называемого итератора (итерируемого множества). Итератор в языках программирования в каждый момент указывает на *текущую позицию* в некоторой линейной упорядоченной последовательности (данных, действий, объектов, ячеек памяти, машинных команд и т.д.). Над итератором можно производить следующие операции:

1. так называемые регулярные операции, в частности, сравнения (на равно, не равно и т.д.);
2. переход к следующему (а иногда и к предыдущему) элементу последовательности. Иногда допускается перемещать итератор сразу на несколько позиций в последовательности;
3. разыменования, при этом через итератор можно получить доступ к текущему элементу. Достум может быть к *значению* текущего элемента (доступ только по чтению) или к *ссылке* на текущий элемент (доступ и по чтению и по записи).

Естественно, переход к следующему (или предыдущему) элементу не определён при достижении конца (начала) последовательности. В теории программирования обычно различают следующие виды итераторов.

Итераторы ввода. Они позволяют проходить последовательность только от начала к концу и только один раз. Два таких итератора нельзя установить на одну последовательность. Типичный пример – последовательность символов в стандартном входном потоке `input`.

Однонаправленные итераторы. Они позволяют проходить последовательность только от начала к концу, но произвольное количество раз, т.е. в любой момент итератор можно установить снова на начало последовательности. Типичные примеры – однонаправленный список и чтение из последовательного файла.

Двунаправленные итераторы. Они поддерживают движение по последовательности как вперёд, так и назад и произвольное количество раз. Типичный пример – двунаправленный список.

Итераторы произвольного доступа. Они позволяют установку на любой элемент последовательности за время, не зависящее от положения этого элемента в последовательности. Ясно, что нельзя установить итератор за границы последовательности. Типичные примеры – одномерный массив и файл прямого доступа.

В простейших случаях итератор реализуется в виде дискретной переменной, далее идут ссылочные переменные, так называемые дескрипторы или более сложные структуры данных, например, объекты.