Блоки кода Clipper [1]

Admin

БЛОКИ КОДА.

Этот раздел охватывает блоки кода. Мы покажем что это такое, как и почему вы используете их. Мы начнем со сравнения блоков кода с макросами, и покажем, когда они могут быть использованы вместо них.

Затем опишем встроенные функции, которые оперируют с блоками, и покажем как блок кода параметризует логику функции. Мы закончим объяснением, как компилировать блок кода во время выполнения программы.

ЧТО ТАКОЕ БЛОК КОДА ?

Блок кода - это тип данных, такой же как дата, логический, числовой, символьный. Блоки имеют имена, как другие переменные, вы можете обозначить их и передать их, как параметры.

Какой тип данных содержит блок? Он просто содержит скомпилированные выражения. Выражения могут быть переданы как параметр, и на некотором этапе вам захочется вычислить их и использовать результат. Это делается, как увидим, с помощью функции eval.

Чтобы создать блок, вы присваиваете его переменной, как вы могли бы предположить. Мы начинаем блок с {|| (выглядит как дорога вдоль морского побережья!) и заканчиваем }. Итак для создания блока rate_eval, при вычислении которого умножаются переменные xchg_rate и price, напишем :

rate_eval = {||xchg_rate * price)

Здесь не вычисляется выражение, это только хранение скомпилированного кода, или блока в переменной rate_eval. Подобно любой другой переменной rate_eval имеет тип ('B') и может быть переназначен, передан как параметр и освобожден. Он может быть вычислен, это будет рассмотрено в следующем разделе.

Важно отметить, что блоки кода обрабатываются компилятором, а не исполняющей системой (как макросы).

Вы можете поставить несколько выражений внутри блока, разделяя их запятыми. Когда вычислится, блок вернет результат последнего выражения, как свое значение.

ВЫЧИСЛЕНИЕ БЛОКА КОДА.

Чтобы вычислить блок, передайте его как параметр в функцию eval, как в

xchg_rate = 2.86
price =100
rate_eval = {||xchg_rate * price)
our_price = eval(rate_eval)

Eval вычисляет переданный блок и возвращает его значение. Конечно, желательно не использовать блоки таким простым способом, но терпение с нами. Блоки являются сильным инструментарием, и чтобы хорошо описать их, мы должны начать с простых примеров.

Следующий блок кода при вычислении вызовет функцию rec() :

{||rec()}

Следующий блок возвращает текущее значение поля tel из текущей базы данных :

{||tel}

Подобно функциям, процедурам и макросам компилятора, блоки могут иметь параметры. То есть, можно определить их в терминах формальных параметров. Затем при вычислении вы заменяете их фактическими параметрами. Вы ставите имена формальных параметров между двумя символами ||. Наши начальные примеры не имели параметров; отметим, что как вызовы функций требуют (), не обращая внимания, принимают ли они параметры, так и блоки требуют ||.

Как пример использования параметров с блоками, рассмотрим выражение для вычисления корня квадратного уравнения (наверное помните борьбу с ними в курсе алгебры высшей школы!) :

(-b+sqrt(b*b-4*a*c))/(2*a)

Как блок, это выглядит так :

{|a,b,c|(-b+sqrt(b*b-4*a*c))/(2*a)}

Блок имеет три параметра : a, Ь, c. Когда он вычисляется, Clipper подставляет фактические параметры вместо формальных. Это похоже на вызов процедуры и функции.

Мы передаем фактические параметры в функцию eval вместе с блоком, как в

```
q_root1={|a,b,c|(-b+sqrt(b*b-4*a*c))/(2*a)}
root1=eval(q_root1,1,5,6)
```

Здесь мы передаем 1 для a, 5 для Ь, 6 для c. Root1 имеет значение -2. Также как в процедурах и функциях, имена формальных параметров - произвольны. Желательно создавать их мнемонику, с учетом того, что они делают.

Вы можете вычислять несколько выражений в блоке кода. Результат равен результату последнего выражения. Например, следующий блок, когда вычисляется, во-первых вызывает функцию test, затем вычисляет переменную i. Возвращаемый результат - новое значение i.

{||test(),++i}

Можно изменять переменную внутри блока кода, просто присваивая ей значение с помощью оператора присваивания:

```
LOCAL var
cb={||var:=...}
```

Отметим, это работает только потому, что можно использовать оператор присваивания внутри выражений.

Самым большим преимуществом блоков кода является параметризация логики подпрограмм. Традиционно вы параметризовали только данные. Вы определяли подпрограмму с формальными параметрами, затем передавали фактические значения при ее вызове. Сейчас вы можете написать подпрограммы, принимающие блоки как параметры. Мы следовательно можем определить, и какие данные использует подпрограмма, что она делает с ними. Мы приведем хорошие примеры в главе 3 с функциями ASCAN ASORT.

БЛОКИ КОДА ПРОТИВ МАКРОСОВ

Блоки кода имеют сходство с макросами. Идея макроса - это хранить выражение, затем позднее вычислять его. Мы храним выражение как символьную строку, как в

mac_1='rec()'

mac_2='tel'

Это не является специальным типом данных. Чтобы вычислить или раскрыть макрос, поставьте символ & перед его именем, как в :

&mac_1

&mac_2

Это выполняет две веши. Во-первых, символьная строка компилируется в выполнимый код. Затем запускается скомпилированный код.

Сопоставим макросы с блоками кода. Блок кода является специальным типом данных, тогда как макрос - это специально сформированная символьная строка. Оба содержат выражения, хотя выражение блока кода обрабатывается компилятором. Макровыражение, с другой стороны, компилируется исполняющей системой. Компиляция - это обычно функция компилятора (!), поэтому чтобы выполнить макросы, Clipper исполняющая система должна содержать мини-компилятор и лексический анализатор. Целый кусок программы.

Каждый раз вычисляя макровыражение. Clipper должен в начале откомпилировать его. Даже если он всегда компилирует в одинаковый код, необходимо повторно вызывать компилятор исполняющей системы - это медленный процесс.

Используя блоки кода, можно отделить компиляцию выражения от его вычисления, Компиляция проводится один раз компилятором, а потом вы можете запускать результат столько раз, сколько пожелаете.

Другая проблема с макросами состоит в том, что нельзя их параметризовать. Для сложения a и b пишем макрос, как 'a+Ь', затем раскрываем его.

Для сложения c и d пишем 'c+d' и т.д. Блоки кода могут быть параметризованы формальными параметрами. Они заменяютсяфактическими параметрами при вычислении блока.

Давайте рассмотрим программу, которая использует макросы, и преобразуем ее для использования блоков. Программа 2-9 показывает простую систему меню, основанную на макросах. Она использует два параллельных массива, один для подсказок меню и другой для имен функций, вызываемых при выборе подсказки. Программа 2-10 показывает ту же программу, написанную с использованием блоков.

```
*Образец программы для иллюстрации макросов. Мы создаем два
*параллельных массива, содержащих подсказки и действия. Actions
*содержит вызовы функций, хранящихся как макросы. Мы используем
*PROMPT/MENU для выдачи подсказки, и если пользователь выберет
*одну из них, то мы активизируем соответствующую функцию.

FUNCTION m_test
LOCAL prompts[3], actions[3],i,choice &amp;amp;&amp;amp; Отметим, local лучше, чем private
Prompts[l] = 'Customers'
prompts[2] = 'Payables'
prompts[3] = 'Receivables'
actions[l] = 'custs()'
actions[2] = 'pays()'
actions[3] = 'recs()'
FOR i=1 TO len(prompts)
@ 1, col()+1 PROMPT prompts[i]
NEXT
choice = 1
MENU TO choice
IF choice &gt; 0
dummy = &amp;amp;(actions[choice])
ENDIF
RETURN NIL &amp;amp;&amp;amp; Не возвращает значения
```

Программа 2-9. Простое меню, использующее макросы

```
*Образец программы для иллюстрации блоков кода. Мы создаем два
*параллельных массива, содержащих подсказки и действия. Actions
*содержит вызовы функций, хранящихся как блоки кода. Мы
*используем PROMPT/MENU для выдачи подсказки, и если
*пользователь выберет одну из них, то мы вычисляем соответствующий
*блок.
FUNCTION m_test
LOCAL prompts[3], actions[3], i, choice
prompts[1] = 'Customers'
prompts[2] = 'Payables'
prompts[3] = 'Receivables'
actionsf[1] = {||custs()}
actions[2] = {||pays()}
actions[3] = {||recs()}
FOR i = 1 TO len(prompts)
@ 1, col() + 1 PROMPT prompts[i]
NEXT
choice = 1
MENU TO choice
IF choice &gt; 0
eval(actions[ choice])
ENDIF
RETURN NIL
```

Программа 2-10. Простое меню, использующее блоки.

Единственное изменение в программе 2-10 - это назначение блока каждому элементу массива и использование eval для вызова функции.

Почему эта программа предпочтительнее? Здесь несколько причин :

*

Каждый раз при выборе элемента меню, в версии с макросом должен компилироваться вызов функции. С блоками компиляция проводится один раз компилятором.

*

Если выражение блока содержит синтаксическую ошибку, то компилятор сообщит об этом. В случае символьной строки, ошибка появится в момент выполнения.

*

Компилятор генерирует внешние ссылки для функций, на которые есть ссылка внутри блока, и компоновщик гарантирует, что они прикомпонованы. Используя макросы, вы должны гарантировать, что функции, на которые вы ссылаетесь, или явно прикомпонованы, или указаны в операторах EXTERNAL.

*

Блоки кода могут вызывать функции STATIC, а макросы нет.

*

Можно компилировать с ключом /w и компилятор будет предупреждать вас о необъявленных именах в блоках кода.

*

Интерпретируя имя, макровыражение предполагает, что это поле, если оно существует. Блок кода следует правилам по умолчанию, как вы и надеялись.

*

Можно поддерживать переменные LOCAL и STATIC внутри блока, даже если они не являются текущими в области видимости.

*

Можно включать несколько выражений в блоки кода; только отделяйте их запятыми

Это простой пример использования блока вместо макроса, и то мы отметили много преимуществ. Реальная сила блока кода проявляется во встроенных функциях, которые изменяются, позволяя программисту передавать необязательный блок.

КОМПИЛЯЦИЯ БЛОКОВ КОДА ВО ВРЕМЯ ВЫПОЛНЕНИЯ

Как мы отмечали, компилятор транслирует блоки кода. Иногда, однако, вы не можете знать содержимое блока до выполнения. Например, пользователь вводит запрос-условие. Он или она вводят строку, такую как

last_name = 'Spence'

и мы храним ее в символьной переменной filt_expr. Для преобразования ее в блок кода мы должны откомпилировать ее во время выполнения.

Предположим filt_expr содержит строку:

{||last_name='Spence'}

Она еще символьная переменная. Для преобразования ее в блок, используем макрооператор, как в

```
filt_expr = '{||last_name = 'Spence'}'
.
.
filt_block = &amp;filt_expr
```

Т.о. можно использовать макрос, чтобы компилировать блоки кода во время выполнения. Компиляция проводится только один раз, когда раскрывается макрос. Впоследствии, при использовании блока он поступает, как любой другой скомпилированный блок.

ЗАМЕЧАНИЕ

Можно использовать макрос для компиляции блока во время выполнения программы Только вы должны гарантировать, что это символьное выражение, начинающееся с {|| и заканчивающееся }.

Блоки не могут хранится в базах данных, но их можно хранить в символьных полях с начальным {|| и конечным } символами. Для компиляции их используйте макрооператор. Например, программа 2-11 снова показывает пример меню, но в этот раз и подсказки и блоки хранятся в записях базы данных. Подсказки хранятся в поле prompt, блоки в поле action.

```
/*Образец программы для иллюстрации блоков кода и компиляции их во время выполнения. Подсказки и
блоки хранятся в базе данных menu в полях prompt, action соответственно. Поле action - символьное, оно
сформировано из начального {11 и конечного }. Мы создаем два параллельных массива из базы данных,
содержащих посказки и действия. Мы 'компилируем' поле action в блок кода перед присваиванием его
элементу массива. Используем PROMPT/MENU для выдачи подсказки, и если пользователь выберет одну из
них, то мы вычисляем соответствующий блок.
*/

FUNCTION m_test

LOCAL prompts[MAX_SIZE], actions[MAX_SIZE], i, choice, num_prompts
USE menu
num_prompts = 0
DO WHILE !eof()
num_prompts++
actions[num_prompts] = &amp;(menu -&gt; action)
```

```
prompts[num_prompts] = menu -&gt; prompt
SKIP
ENDDO
FOR i = 1 TO len(prompts)
@ 1, col() + 1 PROMPT prompts[i]
NEXT
choice = 1
MENU TO choice
IF choice &gt; 0
eval(actions[choice])
ENDIF
RETURN NIL
```

Программа 2-11. Memo, управляемое базой данных.

Отметим, что программа 2-11 компилирует блок непосредственно иэ базы данных :

```
actions[num_prompts] = &amp;(menu -&gt; action)
```

Мы предполагаем, что поле action сформировано как блок кода. Т.е. оно имеет начальные символы {|| и конечный}. В общем, если иметь выражение в символьной переменной c_var, то можно его преобразовать в блок так

```
&amp;('{||' + c_var + '}')
```

Напишем преобразование, как макрос компилятора следующим образом :

```
#define COMPILE(c_expr) &amp;('{||' + c_expr + '}')
```

Предположим, поля action содержат просто вызовы функций, как

```
Custs()
pays()
recs()
```

Т.е. они не сформированы как блоки. Мы напишем цикл построения массива так :

```
num_prompts = 0
DO WHILE !eof()
num_prompts++
actions[num_prompts] = COMPILE(menu -&gt; action)
prompts[num_prompts] = menu -&gt; prompt
SKIP
ENDDO
```

Сейчас, зная, как компилировать блоки кода во время выполнения, мы можем использовать их для замены неэффективного кода. Типичное использование макроса - это позволить пользователю определять фильтр-условие в момент выполнения программы. Он или она определяют его, как символьное выражение и после проверки его на правильность, мы используем его как фильтр для записей из базы данных. Например, принимаемая переменная while_scope содержит символьную строку

```
parts -&gt; part_num = this_part
```

Желательно обрабатывать базу данных подобно этому :

```
DO WHILE &amp;while_scope
SKIP
ENDDO
```

Вспомним, каждый раз раскрывая макрос, Clipper делает две веши - компилирует макрос, затем запускает его. Очевидно, повторное компилирование символьной строки является тратой времени и расточительством. Она всегда компилируется в одинаковый код, но с макросами мы не имеем способа сохранения результата для дальнейшего их использования.

С блоками кода мы можем делать это точно. Можно переписать предыдущую программу, как

```
cb = COMPILE(while_scope)
DO WHILE eval(cb)
.
.
SKIP
ENDDO
```

Мы отделили компиляцию от выполнения.

Общее требование - преобразовать символьную переменную, содержащую имя поля в блок кода, который возвращает его значение. При использовании системы TBROWSE (Обсуждаемой в главе 7) мы определяем поля, которые хотим просмотреть, как блоки кода. Для осуществления этого намерения требуется способ преобразования символьной временной, содержащей имя поля, в блок кода.

Как известно, это можно сделать так.

```
field_name = 'L_name' // поле базы данных
field_blk = &amp;('{||' + field_name + '}')
```

Проблема здесь в том, что мы должны прикомпоновать программу обработки макроса из Clipper.lib. Кроме того, как известно, макрасширение работает медленно.

Признавая это, Nantucket (благослави господи его корпоративную душу!) внедрил функции, возвращающие блок кода для доступа к полю и для его изменения. Они позволяют строить блок кода во время выполнения без макрооператора.

Существуют две функции fieldblock и fieldwblock. Обе принимают имя поля как параметр, и обе возвращают блок кода для доступа к полю. Различие в том, что fieldwblock Устанавливает номер фиксированной рабочей области в блок, который возвращает, этому он может быть вычислен, не обращая внимания, отселектирована ли база данных в настоящий момент.

Используя fieldblock, перепишем предыдущий пример так

```
field_blk = fieldblock(field_name)
```

Тип field_blk - это B для блока. Когда мы вычисляем его, используя функцию eval, он Возвращает текущее значение поля L_name.

Блок кода, возвращаемый из этих функций, это то, что Nantucket называет блок get/set, Мы показали, как использовать его для доступа к полю, но если передавать второй параметр в eval (т.е. параметр в блок кода), он изменит поле на его значение. Например,

```
eval(field_blk, 'Spence')
```

Изменит значение поля L_name на 'Spence'.

Чтобы понять, как блок кода делает это, рассмотрим, как можно написать то же самое помощью макрооператора

```
bsg = &amp;('{|setval| iif(setval = NIL, L_name,;
FIELD-&gt;L_name := setval)}')
```

При передаче параметра в блок, поле изменено на его значение. В противном случае его значение остается прежним. Это тоже самое, что делает блок, возвращенный из fieldblock.

Наконец, существует функция, которая делает то же самое с переменными памяти, называемая memvarblock. Передаете ей имя переменной и функция возвращает блок get/ set для нее.

СОВЕТ

Используйте функции fieldblock, fieldwblock и memvarblock вместо макрооператора для создания блоков get/set для полей и переменных.

LOCAL И STATIC В БЛОКАХ.

Если вы должны создать и использовать переменную LOCAL внутри блока, обьявите ее как параметр. Помешайте его в конце списка параметров, чтобы он не смешивался с действительными параметрами. Если он не будет передан, вы можете использовать его, и Clipper создаст для него эквивалент переменной local.

Переменная имеет область действия только внутри блока, поэтому, как только он закончится, имя переменной и ее значение исчезнут.

Можно использовать блоки кода для доступа к переменным local и static, которые иначе могут быть не доступны. Доступ к ним осуществляется внутри блока и всякий раз, когда блок вычисляется, он ссылается к переменным в той области видимости, где он был создан.

Можно также использовать блоки для вызова подпрограмм static, которые иначе могут быть не доступны. Например, предположим, что вы пишете в одном программном файле:

```
STATIC FUNCTION initializer
RETURN
FUNCTION opener
LOCAL cb := {||initializer()}
anotherfunc(cb)
RETURN
```

и в другом :

```
FUNCTION anotherfunc(code_block)
eval(code_block)
RETURN
```

Когда anotherfunc вычисляет code_block, он вызывает static функцию initializer. Если бы она попыталась вызвать initializer непосредственно, возникла бы ошибка компоновщика, т.к. не существует определения initializer во время компоновки.

РЕЗЮМЕ О БЛОКАХ КОДА.

В этом разделе рассмотрены блоки кода. Блок является типом данных, который хранит скомпилированный код. Мы показали, как блоки заменяют макросы, но основное преимущество в использовании их - это параметризация операторов подпрограмм.

Тип материала: Практикум [2]

Источник (modified on 31/10/2014 - 09:56): https://www.spb4plus.ru/praktikum/bloki-koda-clipper

Ссылки
[1] https://www.spb4plus.ru/praktikum/bloki-koda-clipper
[2] https://www.spb4plus.ru/kategoriya/praktika