Тема 1.9 Рекуррентные выражения. Рекурсия прямая и косвенная

Программы, которые мы обсуждали, имели в основном структуру функций, которые вызывают другие функции, подчиняясь строгой иерархии. Для некоторых типов задач полезно иметь функции, которые вызываю сами себя.

Рекурсивная функция – это функция, которая вызывает саму себя или непосредственно или косвенно через другую функцию.

Рекурсия – такой способ организации функции, при котором эта функция в ходе выполнения ее операторов обращается сама к себе.

Рекурсивные подходы к решению задач имеют ряд общих элементов. Для решения задачи вызывается рекурсивная функция. Фактически функция знает решение только простейшего случая, или так называемого основного случая. Если функция вызывается для решения основного случая, то она просто возвращает результат. Если функция вызывается для решения более сложной проблемы, функция делит задачу на две обобщенные части: часть, для которой функция имеет способ решения, и часть, для которой функция не имеет решения. Чтобы сделать рекурсию возможной, последняя часть должна быть похожа на первоначальную, функция запускает свою новую копию, чтобы продолжить решение меньшей задачи. Этот процесс называется рекурсивным вызовом или шагом рекурсии. Шаг рекурсии также включает ключевое слово return, поскольку ее результат будет объединен с предыдущей частью задачи, для которой функция знала решение, чтобы сформировать окончательный результат, который будет передан первоначальной вызывающей функции.

В то время как выполняется шаг рекурсии, первоначальное обращение к функции остается открытым, т.е. его выполнение не завершается. Шаг рекурсии может приводить к намного большему числу таких рекурсивных вызовов, которые продолжаются до тех пор, пока функция продолжает делить каждую последующую задачу, для решения которой она вызывалась, на две обобщенные части. Чтобы рекурсия завершилась, функция каждый раз должна вызываться для решения все более простого варианта первоначальной задачи, и эта последовательность все меньших и меньших задач должна, в конце концов, свестись к основному случаю. В этой точке функция распознает основной случай, возвращает результат предыдущей копии функции, и далее следует последовательность возвратов по обратному пути до тех пор, пока первоначальный вызов функции не вернет конечный результат в main.

В рекурсивном определении должно присутствовать ограничение, граничное условие, при выходе на которое дальнейшая инициация рекурсивных обращений прекращается.

Например: определение факториала

$$n!=1*2*3*\ldots*n$$

$$n!=\begin{cases} 1, \text{если } n <= 1, \\ (n-1)!*n, \text{если } n > 1 \end{cases}$$

Факториал целого числа может быть вычислен итеративно (не рекурсивно) с помощью цикла for:

```
Factorial=1;
For (counter=number; counter>=1; counter--)
        Factorial*=counter;
```

```
double Factorial(int n)
{
        double f;
        if (n <= 1)
                        f=1;
                else
                        f=Factorial(n-1)*n;
        return f;
}
```

Пример 2: Вычисление суммы элементов линейного массива

При решении задачи используем следующее:
- сумма равна нулю, если количество элементов равно нулю;
- сумма равна сумме всех предыдущих элементов плюс последний, если количество элементов не равно нулю.

```
# include <stdio.h>
# include <conio.h>
# include <stdlib.h>
# include <time.h>
int summa(int n, int a[]);
int i,n,a[100];
void main()
{
clrscr();
printf("Количество элементов массива?");
scanf("%d", &n);
printf("В массиве %d элементов", n);
randomize();
for (i=0; i<n; i++)
{ a[i]=-10+random(21); printf("%d", a[i]);}
        printf("Сумма: %d", summa(n-1,a));
}

int summa(int n, int a[100])
{
        if (n==0)
          return a[0];
        else
          return a[n]+summa(n-1,a);
}
```

Пример3: Рассмотрим ряд Фибоначчи:

1,1,2,3,5,8,13,21,34,.,

Каждое число ряда (после второго) представляет собой сумму двух стоящих впереди чисел. Задача может состоять в том, чтобы, например, определить 12-й член ряда Фибоначчи.

Один из способов решения этой проблемы лежит в тщательном анализе этого ряда. Первые два числа равны 1. Каждое последующее число равно сумме двух предыдущих. Таким образом, семнадцатое число равно сумме шестнадцатого и пятнадцатого. В общем случае /1-е число равно сумме (n-2)-го и (n-1)-то при условии, если n > 2.

Для рекурсивных функций необходимо задать условие прекращения рекурсии. Обязательно должно произойти нечто, способное заставить программу остановить

рекурсию, или же она никогда не закончится. В ряду Фибоначчи условием останова является выражение n < 3.

При этом используется следующий алгоритм:

1. Предлагаем пользователю указать, какой член в ряду Фибоначчи следует рассчитать.

2. Вызываем функцию fib(), передавая в качестве аргумента порядковый номер члена ряда Фибоначчи, заданный пользователем.

3. В функции fib() выполняется анализ аргумента (n). Если n < 3, функция возвращает значение 1; в противном случае функция fib() вызывает сама себя (рекурсивно), передавая в качестве аргумента значение n-2, затем снова вызывает самое себя, передавая в качестве аргумента значение n-1, а после этого возвращает сумму.

Если вызвать функцию fib(1), она возвратит 1. Если вызвать функцию fib(2), она также возвратит 1. Если вызвать функцию fib(3), она возвратит сумму значений, возвращаемых функциями fib(2) и fib(1). Поскольку вызов функции fib(2) возвращает значение 1 и вызов функции fib(1) возвращает значение 1,то функция fib(3) возвратит значение 2.

Если вызвать функцию fib(4), она возвратит сумму значений, возвращаемых функциями fib(3) и fib(2). Мы уже установили, что функция fib(3) возвращает значение 2 (путем вызова функций fib(2) и fib(1)) и что функция fib(2) возвращает значение 1, поэтому функция fib(4) просуммирует эти числа и возвратит значение 3, которое будет являться четвертым членом ряда Фибоначчи.

Сделаем еще один шаг. Если вызвать функцию fib(5), она вернет сумму значений, возвращаемых функциями fib(4) и fib(3). Как мы установили, функция fib(4) возвращает значение 3, а функция fib(3) — значение 2, поэтому возвращаемая сумма будет равна числу 5.

Описанный метод — не самый эффективный способ решения этой задачи (при вызове функции fib(20) функция fib() вызывается 13 529 раз!), тем не менее, он работает. Однако будьте осторожны. Если задать слишком большой номер члена ряда Фибоначчи, вам может не хватить памяти. При каждом вызове функции fib() резервируется некоторая область памяти. При возвращении из функции память освобождается. Но при рекурсивных вызовах резервируются все новые области памяти, а при таком подходе системная память может исчерпаться довольно быстро. Реализация функции fib() показана в листинге.

Пример использования рекурсии для нахождения члена ряда Фибоначчи

```
1: #include <iostream.h>
2:
3: int fib (int n);
4:
5: int main()
6: {
7:
8: int n, answer;
9: cout << "Enter number to find: ";
10: cin >> n;
11: cout << n\ n";
12:
13: answer = fib(n);
14:
15: cout << answer << " is the " << n << "th Fibonacci number\ n";
16: return 0;
```

```
17: }
18: int fib (int n)
19: {
20: cout << "Processing fib(" << n << ")... ";
21: if (n < 3 )
22: {
23:    cout << "Return 1!\ n";
24:    return (1);
25: }
26: else
27: {
28:    cout << "Call fib(" << n-2 << ") and fib(" « n-1 << ").\ n";
29:    return( fib(n-2) + fib(n-D);
30: }
31: }
```

Результат:

```
Enter number to find: 6Processing fib(6). . . Call fib(4) and fib(5)
Processing fib(4).. . Call fib(2) and fib(3)
Processing fib(2).. . Return 1!
Processing fib(3).. . Call fib(l) and fib(2)
Processing fib(l),. . Return 1!
Processing fib(2).. . Return 1!
Processing fib(5).. . Call fib(3) and f i b (4)
Processing fib(3).. . Call fib(1) and fib(2)
Processing fib(1).. . Return 1!
Processing fib(2).. . Return 1!
Processing fib(4).. . Call fib(2) and fib(3)
Processing fib(2).. . Return 1!
Processing f i b (3). . . Call fib(1) and fib(2)
Processing fib(i).. . Return 1!
Processing fib(2).. . Return 1!
8 is the 6th Fibonacci number
```

В строке 9 программа предлагает ввести номер искомого члена ряда и присваивает его переменной n. Затем вызывается функция fib() с аргументом n. Выполнение программы переходит к функции fib(), где в строке 20 этот аргумент выводится на экран.

В строке 21 проверяется, не меньше ли аргумент числа 3, и, если это так, функция fib() возвращает значение 1. В противном случае выводится сумма значений, возвращаемых при вызове функции fib() с аргументами n-2 и n-1. Таким образом, эту программу можно представить как циклический вызов функции fib(), повторяющийся до тех пор, пока при очередном вызове этой функции не будет возвращено некоторое значение. Единственными вызовами, которые немедленно возвращают значения, являются вызовы функций fib(1) и fib(2). Рекурсивное использование функции fib() проиллюстрировано на рис. 6.4 и 6.5.

В примере, изображенном на рисунках, переменная n равна значению 6, поэтому из функции main() вызывается функция fib(6). Выполнение программы переходит в тело функции fib(), и в строке 30 значение переданного аргумента сравнивается с числом 3. Поскольку число 6 больше числа 3, функция fib(6) возвращает сумму значений, возвращаемых функциями fib(4) и fib(5):

```
38: return( fib(n-2) + fib(n-1));
```

Это означает, что выполняется обращение к функциям fib(4) и fib(5) (поскольку переменная n равна числу 6, то fib(n-2) — это то же самое, что fib(4), а fib(n-1) — то же самое, что fib(5)). После этого функция fib(6), которой в текущий момент передано управление программой, ожидает, пока сделанные вызовы не возвратят какое-нибудь

значение. Дождавшись возврата значений, эта функция возвратит результат суммирования этих двух значений.

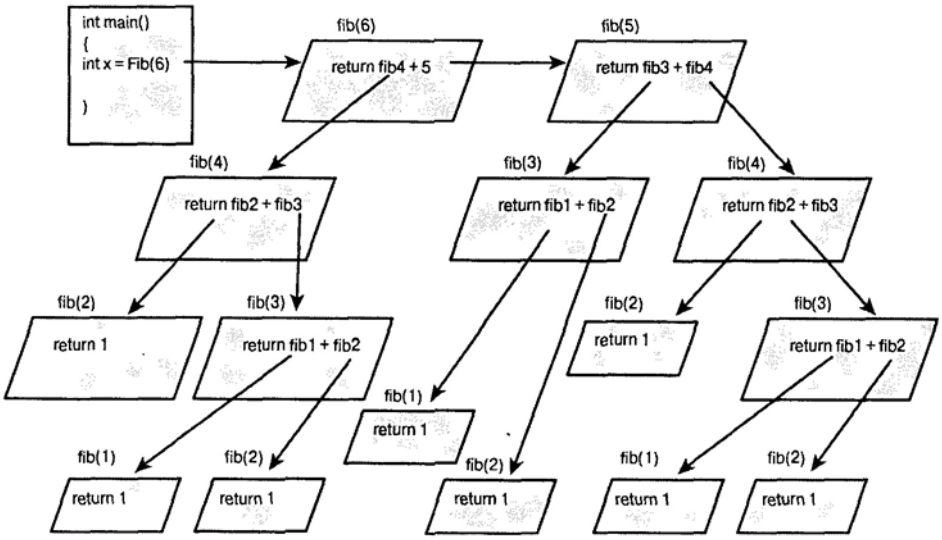

Рисунок 6.5 - Использование рекурсии

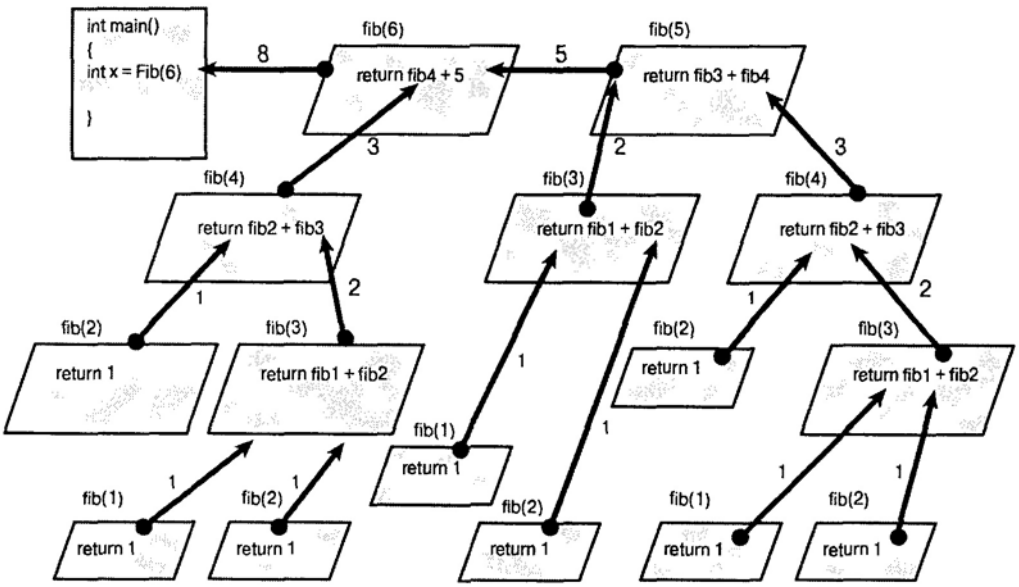

Рисунок 6.6 - Возвращение из рекурсии

Поскольку при вызове функции fib(5) передается аргумент, который не меньше числа 3, функция fib() будет вызываться снова, на этот раз с аргументами 4 и 3. А функция flb(4) вызовет, в свою очередь, функции fib(3) и fib(2).

Результаты и промежуточные этапы работы программы, представленной в листинге 6.4, выводятся на экран. Скомпилируйте, скомпонуйте и выполните эту программу, введя сначала число 1, затем 2, 3, и так доберитесь до числа 6, внимательно наблюдая за отображаемой информацией.

Работа с этой программой предоставляет вам прекрасный шанс проверить возможности своего отладчика. Разместите точку останова в строке 20, а затем заходите в тело каждой вызываемой функции fib(), отслеживая значение переменной n при каждом рекурсивном вызове функции fib().

В программировании на языке C++ рекурсия не частый гость, но в определенных случаях она является мощным и весьма элегантным инструментом.

Директива препроцессора # define: макросы

Макрос, определяемый директивой препроцессора #define, это символическое имя некоторых операций. Идентификатор макроса заменяется на замещающий текст до начала компиляции программы.

Макросы могут быть определены с параметрами или без них. Если макрос имеет параметры, то сначала в замещающий текст подставляются значения параметров, а затем уже этот расширенный макрос подставляется в текст вместо идентификатора макроса и списка его параметров.

Рассмотрим следующий макрос с одним параметром для расчета площади круга:

```
#define   CIRCLE_AREA(x) (PI*(x)*(x))
```

Везде в файле, где появится идентификатор CIRCLE_AREA(x), значение аргумента x будет использовано для замены x в замещающем тексте, символическая константа PI будет заменена ее значением (определенным выше) и этот расширенный текст макроса будет использован для замещения.

Например, оператор с макросом в тексте программы:

```
area  =  CIRCLE_AREA(4);
```

примет вид:

```
area  =   (3.14159*(4)*(4));
```

Поскольку это выражение состоит только из констант, его значение будет вычислено во время компиляции и полученный результат будет присвоен переменной area во время выполнения программы. Круглые скобки вокруг каждого включения параметра x в тексте макроса и вокруг всего выражения применяются для того, чтобы обеспечить соответствующий порядок вычислений в том случае, когда аргументом макроса является выражение.

Например, при раскрытии макроса в операторе:

```
area = CIRCLE_AREA(c + 2);
```

оператор примет следующий вид:

```
area  =   (3.14159*(c  +  2) * (c   +  2));
```

и вычисления будут выполняться правильно, потому что скобки обеспечили правильную последовательность вычислений.

Если в определении макроса опустить все круглые скобки, то после расширения макроса рассматриваемый оператор будет иметь следующий вид:

```
area  =  3.14159*c   +  2*c  +  2;
```

С учетом приоритетов операций, вычисление значения выражения будет сделано в следующем порядке

```
area  =   (3.14159*c) + (2*c) +2;
```

и будет получен неверный результат.

Вычисления, выполняемые макросом CIRCLE_AREA, можно выполнять при помощи функции.

Функция circleArea:

```
double   circleArea(double  x)    { return   3.14159 * x * x;}
```

выполняет те же вычисления, что и макрос CIRCLE_AREA.

Недостатком использования этой функции является то, что на ее вызов должны быть затрачены некоторые ресурсы.

Преимущества использования макроса CIRCLE_AREA состоят в том, что вычисления, выполняемые макросом, непосредственно помещаются в текст программы и не приводят к дополнительным накладным расходам, связанным с вызовом функции; при этом текст программы остается легко читаемым, потому что имя CIRCLE_AREA говорит само за себя.

Недостаток этого макроса в том, что значение аргумента вычисляется дважды. И конечно, макрос будет расширяться при каждом своем появлении в тексте программы. Если макрос большой, то это приводит к увеличению размера программы. Таким образом, нужно искать компромисс между быстродействием и размером программы.

Определения символических констант и макросов могут быть аннулированы при помощи директивы препроцессора #undef, Директива #undef отменяет определение симнолической константы или макроса. Область действия символической константы или макроса начинается с места их определения и заканчивается явным, их аннулированием директивой #undef или концом файла. После аннулирования соответствующий идентификатор может быть снова использован в директиве #define.

Условная компиляция

Условная компиляция дает возможность программисту управлять выполнением директив препроцессора и компиляцией программного кода. *Условные конструкции* препроцессора позволяют компилировать или пропускать часть программы в зависимости от выполнения некоторого условия.

Каждая условная директива препроцессора вычисляет значение целочисленного константного выражения. Операции преобразования типов, операция sizeof и константы перечислимого типа не могут участвовать в выражениях, вычисляемых в директивах препроцессора.

Условие может принимать одну из описываемых ниже форм.

#if константное_выражение

Проверяется значение выражения, составленного из констант и если оно не равно нулю, компилируется (включается) последующий текст.

#ifdef идентификатор

Последующий тест компилируется, если "идентификатор" уже был определен для препроцессора в команде #define

#ifndef идентификатор

Последующий текст компилируется, если "идентификатор" в данный момент не определен. Конструкция

#undef идентификатор

исключает "идентификатор" из списка определенных для препроцессора имен. За любой из трех условных команд может следовать произвольное число строк текста, содержащих, возможно, команду вида #else и заканчивающихся #endif. Если проверяемое условие справедливо, то строки между #else и #endif игнорируются. Если же проверяемое условие не выполняется, то игнорируются все строки между проверкой и командой #else, а если ее нет, то командой #endif.

Условная директива препроцессора во многом похожа на оператор if. Рассмотрим следующий фрагмент кода:

```
#if   !defined(NULL)
#define  NULL  0
#endif
```

Эти директивы определяют, не была ли определена ранее константа NULL. Выражение defined(NULL) дает значение 1, если NULL определена, и 0 в противном случае.

Если результат равен 0, то выражение !defined(NULL) дает значение 1 и в следующей строке производится определение константы NULL. В противном случае директива #define пропускается.

Каждая директива #if должна заканчиваться своим #endif.

Директивы #ifdef и #ifndef являются сокращением выражений #if defined(uMM) и #if \defined(uMH).

Можно использовать сложные конструкции условных директив препроцессора при помощи директив #elif (эквивалент else if в структуре if) и #else (эквивалент else в структуре if).

При разработке программы программисты часто находят удобным для себя временно «закомментировать» большие фрагменты кода и не компилировать их. Если в коде используются комментарии в стиле C, то знаки комментария /* и */ не помогут решить эту задачу. В таком случае программист может использовать следующую конструкцию директив препроцессора

#if 0

Фрагмент кода, который не нужно компилировать #endif

Для того, чтобы этот фрагмент включить в процесс компиляции, достаточно заменить 0 в приведенной конструкции на 1.

Приведенная на примере программа иллюстрирует применение некоторых из рассмотренных выше команд, обеспечивающих условную компиляцию.

Листинг. Пример использования условной компиляции

```
1: #define SIZE 16
2: #include stdio.h
3: main()
4: {
5:    char c = 'A';
6:    #ifdef SIZE
7:       int x = 123;
8:       printf("x=%d\n",x);
9:    #else
10:      static char x[SIZE] = "информатика";
11:      printf("x=%s\n",x);
12:   #endif
13:   printf("%c\n",c);
14: }
```

Условная компиляция обычно используется как средство отладки. Многие системы программирования на C++ предоставляют разработчику отладчики программ. Однако, сначала нужно изучить этот отладчик и научиться его использовать, что часто вызывает затруднения у студентов и начинающих программистов. Вместо отладчика можно использовать операторы вывода значений переменных, что позволяет контролировать процесс выполнения программы. Эти операторы «обкладываются» условными директивами препроцессора и компилируются только пока процесс отладки программы не завершен.

Например, в следующем фрагменте:

```
#ifdef DEBUG
cout « "Переменная x = " « x « endl;
#endif
```

оператор вывода в поток cout будет компилироваться только в случае, если символическая константа DEBUG была определена (директивой #define DEBUG) до директивы #ifdef DEBUG. После завершения процесса отладки директива #define DEBUG может быть просто удалена из исходного файла и операторы вывода, нужные только для целей отладки, будут игнорироваться во время компиляции. В больших программах, возможно, потребуется определять несколько различных символических констант, которые могут управлять условной компиляцией различных частей исходного файла