Абстрактные типы данных. Стек вызовов, рекурсия.

А. Г. Фенстер, `fenster@fenster.name`

10 апреля 2010 г.

На этом занятии мы поговорим о применении и способах реализации стека, очереди и кучи. Также будет разобрано, что происходит при рекурсивном вызове функции и как можно избавиться от рекурсии, применяя вместо неё работу со стеком.

1 Структуры данных

Информация о стеках, очередях, деках и кучах представлена в отдельном файле.

2 Рекурсивные функции и работа со стеком

Т. к. функции могут вызывать другие функции, в каждый момент времени работы программы набор выполняемых функций можно представить в виде стека: например, если функция `main` вызвала функцию `f`, а функция `f` вызвала функцию `g`, то на верхушке стека будет `g`, «под» ней — `f` и затем `main`. Такой стек поддерживается при работе программы, написанной практически на любом современном императивном языке, и называется *стеком вызовов* (call stack). Элемент стека вызовов, *стековый кадр* (stack frame), содержит информацию об одном запущенном экземпляре функции.

Не углубляясь в низкоуровневые детали, обсудим подробнее содержимое стекового кадра. Пусть функция `f` вызывает функцию `g`. Когда выполнение функции `g` завершится, управление должно вернуться

в функцию `f` в точку, **следующую за вызовом** `g`. При этом должен восстановиться контекст, т. е. значения всех переменных функции `f` (как локальных переменных, так и параметров).

Соответственно, в стековом кадре сохраняются как минимум следующие данные:

1. адрес возврата: точка в коде, куда должно передаться управление после завершения работы функции;
2. значения параметров функции;
3. значения локальных переменных функции.

Точный формат и содержимое стекового кадра зависят от языка, компилятора и архитектуры компьютера и на данном этапе нас не интересуют.

Понимание того, что должно содержаться в стеке вызовов, позволяет легко переписать любую рекурсивную функцию без использования рекурсии, но с использованием стека. Сначала мы разберём простой пример, затем сформулируем абсолютно формальный алгоритм, пользуясь которым, можно избавиться от рекурсии в любой функции.

Рассмотрим рекурсивную функцию префиксного (прямого) обхода бинарного дерева:

```
void prefix(struct node *root)
{
    if (!root)
        return;
    printf("%d\n", root->data);
    prefix(root->left);
    /* 1-я точка возврата из рекурсии */
    prefix(root->right);
    /* 2-я точка возврата из рекурсии */
}
```

Если вызвать эту функцию для корня бинарного дерева, она будет рекурсивно вызвана для каждого узла дерева. Кроме того, будет сделано несколько «лишних» вызовов (с нулевым параметром) в тех случаях, когда у текущего узла нет одного или двух сыновей.

Функция `prefix`, хоть и не является хвостовой рекурсией, весьма проста: глядя на её код, понятно, что если первый рекурсивный вызов (левая ветвь) закончен (1-я точка возврата), функция сразу же вызывает себя для правой ветви, а после второго рекурсивного вызова сразу же

происходит выход из функции. Если задаться целью переписать такой обход без рекурсии, можно хранить текущий узел в переменной (`struct node *curr`), а рекурсивные вызовы заменить присваиваниями переменной `curr`. Но как добиться «разветвления» функции? В данном конкретном случае достаточно всего лишь сохранять в стеке адрес пока ещё не посещённой правой вершины, уходя из текущей вершины в левую, чтобы вернуться к нему, когда левое поддерево будет посещено.

Следующая нерекурсивная функция, работающая со стеком, хранящим значения типа `struct node *`, является аналогом функции `prefix`:

```
void prefix_no_recursion(struct node *curr)
{
    while (curr || !empty())
    {
        if (!curr)
            curr = pop();
        printf("%d ", curr->data);
        if (curr->right)
            push(curr->right);
        curr = curr->left;
    }
}
```

Функция меняет значение своего параметра `curr`, но это не должно вас смущать: меняется именно `curr`, а не `*curr`, так что побочных эффектов здесь не возникает, а `curr` используется просто как локальная переменная.

Прежде чем читать дальше, пожалуйста, потратьте несколько минут на «ручную трассировку» (на бумаге) приведённых выше функций.

Ещё раз обращу внимание на то, что функция `prefix` очень простая: при возврате из рекурсивного вызова из всего текущего узла алгоритму требуется знать лишь адрес правого сына, поэтому в нерекурсивной версии функции мы вместо адреса возврата, как положено, сохраняем в стеке именно адрес правого сына, сильно упрощая тем самым логику работы функции. По сути дела, мы не переписали функцию `prefix` без рекурсии, а написали похожую функцию, но с несколько другой логикой. Если требуется сделать «честный» откат и хранить именно адрес точки возврата (а не адрес точки, в которую мы прыгнем через шаг), нерекурсивная реализация могла бы быть такой:

```
void prefix_no_recursion_2(struct node *curr)
{
    while (curr || !empty())
    {
```

```
        if (!curr)
        {
            curr = pop();
            curr = curr->right;
        }
        if (curr)
        {
            printf("%d ", curr->data);
            push(curr); /* сохранили узел, в который вернёмся */
            curr = curr->left;
        }
    }
}
```

Этот вариант уже намного более похож на исходную рекурсивную функцию.

Тем не менее, при написании обоих нерекурсивных вариантов функции `prefix` мы активно пользовались пониманием того, что эта функция делает. На самом деле для переписывания рекурсивной функции на работу со стеком полного понимания устройства функции зачастую не требуется. Если внимательно посмотреть на исходную рекурсивную функцию и отметить у неё все точки выхода и рекурсивные вызовы, создать нерекурсивную версию можно совершенно механическим способом.

Посмотрим на код функции `prefix` внимательнее и пронумеруем основные точки:

```
void prefix(struct node *root)
{
    /* точка 0 */
    if (!root)
        return; /* выход из функции в точку возврата */
    printf("%d\n", root->data);
    prefix(root->left);  /* рекурсивный вызов */
    /*---------------------------------------*/

    /* точка 1 */
    prefix(root->right); /* рекурсивный вызов */
    /*---------------------------------------*/

    /* точка 2 */
    /* выход из функции в точку возврата */
}
```

Пусть наш стек умеет сохранять пары вида `(int, struct node *)`: число будет номером точки возврата (0, 1 или 2 в коде выше), а указатель — значением параметра `root`. Соответственно, будем считать, что `push` принимает два параметра и кладёт в стек полученную пару, а `pop` возвращает значения элементов пары опять же через свои параметры.

Создадим в нашей функции бесконечный цикл и две локальных переменных: `position` и `curr` — текущая позиция (0, 1 или 2) и адрес текущей вершины дерева. А теперь совершенно механически перепишем код:

```
void prefix(struct node *curr)
{
    int position = 0;

    for ( ; ; )
    {
        if (position == 0)
        {
            /* точка 0 */
            if (!curr)
            {
                if (empty()) break;
                pop(&position, &curr);
                continue;
                /* выход из функции в точку возврата */
            }
            push(1, curr); /* запомнили точку возврата */
            position = 0;
            curr = curr->left;
            continue; /* рекурсивный вызов */
        }
        else if (position == 1)
        {
            push(2, curr); /* запомнили точку возврата */
            position = 0;
            curr = curr->right;
            continue; /* рекурсивный вызов */
        }
        else if (position == 2)
        {
            if (empty()) break;
            pop(&position, &curr);
            continue;
            /* выход из функции в точку возврата */
        }
    }
}
```

Сформулируем простые правила переписывания:

1. в местах рекурсивных вызовов запоминаем в стеке позицию возврата и текущие значения всех переменных, выставляем новые значения параметров и переходим в начальную позицию функции;

2. в местах выхода из функции восстанавливаем текущие значения всех переменных из стека и переходим в позицию, также сохранённую в стеке. Если же стек пуст, завершаем выполнение функции.