ЛЕКЦИЯ 8

СТРУКТУРЫ ДАННЫХ. СТЕК

На этой лекции речь пойдёт о структурах данных. Теперь, когда синтаксис языка хорошо изучен, можно приступать к более сложным вещам.

Задача Дана строка, состоящая из открывающих и закрывающих скобок, например, (()()). Количество открывающих скобок равно количеству закрывающих. **Правильной структурой** называется такая последовательность скобок, в которой скобки формируют непересекающиеся вложенные пары. Требуется определить, является ли данная скобочная структура правильной. *

Эта задача решалась так. Заводился счётчик и производился цикл по элементам строки. Каждая открывающая скобка увеличивала его на единицу, а каждая закрывающая — уменьшала его на единицу. Условия правильности были такие: в конце счётчик должен обнулиться, и на любой итерации он должен быть неотрицателен.

Задача Теперь видоизменим задачу 8.1, введя два вида скобок — круглые и треугольные. Строка теперь может выглядеть, например, как (<>()). Теперь нужно распознавать структуру типа (<)> как неправильную. *

В этой задаче счётчик применять бесполезно, потому что нужно отличать, какой открывающей скобке соответствует закрывающая. Значит, нужно применять другой метод.

Решение.

Организуем очередь из символов, и пройдёмся в цикле по элементам исходной строки. Доступ к этой очереди происходит не с начала, а с её конца. Если на какой-то итерации обнаружится, что очередь пуста, а очередная скобка — закрывающая, то нужно также прервать выполнение алгоритма и выдать, что структура неправильная.

1. Если очередным элементом является открывающая скобка, то помещаем её в очередь.

2. Если же это закрывающая скобка, то нужно сравнить её тип с типом последней скобки в очереди.

(a) Если очередь пуста, то нужно прервать выполнение алгоритма и выдать, что структура неправильная.

(b) Если эти типы совпадают, то удаляем последнюю скобку из очереди.

(c) Если же типы не совпадают, то скобочная структура не является правильной, и алгоритм завершается.

Если алгоритм проработал до конца, и после цикла очередь пуста, то скобочная структура является правильной.

Добавление в задачу ещё одного типа скобок, например, квадратных скобок, не меняет метода решения.

Тип доступа к последнему элементу называется **FILO**: First In — Last Out (первым вошёл — последним обслужен). Такой тип доступа к данным имеет структура данных **стек**. Очередь в столовой, например, имеет другой тип доступа. Он называется **FIFO**: First In — First Out (первым вошёл — первым обслужен), а структура данных, использующая такой тип доступа, называется **очередью**.

1. Интерфейсы и реализация

Представим себе часы как элемент программы. У них есть следующие интерфейсы:

1. «посмотреть время»;

2. «завести будильник».

Функция показа времени может быть реализована по-разному, в зависимости от типа часов. Если часы механические, то показ времени осуществляется с помощью стрелок. Если часы электронные, то это могут быть либо стрелки, либо цифры. Если часы солнечные, то они не смогут показывать минуты и секунды, но посмотреть, сколько они показывают часов, по-прежнему можно (когда на них падают солнечные лучи). Таким образом, интерфейс тот же самый — «посмотреть время», а реализация — совершенно разная.

На данной лекции будут описаны интерфейсы FILO (стек) и FIFO (очередь).

Рис. 3.1

2. Стек

Приведём аналогию из детства. Пирамидки, надевающиеся на палочку, снимаются с неё в обратном порядке: сначала снимается кольцо, надетое последним, затем — предпоследним, и т. д. (рис. **??**). Также стек можно представлять как стопку тарелок, или стопку книг, из которой в каждый момент можно достать только верхнюю из них.

Есть устоявшиеся названия для операций со стеком. Интерфейс стека содержит всего две операции.

1. Операция «Push» — положить элемент в стек.
2. Операция «Pop» — достать элемент из стека.

Также иногда полезна операция «Is_empty», возвращающая true, если в стеке есть элементы, и false в противном случае. Рисунок **??** схематически показывает, как в стек кладутся элементы.

Рассмотрим ещё одну задачу, которая решается с помощью стека. Возьмём обычную арифметическую запись, в которой есть числа и бинарные операции $+$, $-$, $*$, ¥. Например,

$$2 * 3 + 4 * 5. \tag{8.1}$$

Это так называемая **инфиксная** запись — операторы находятся между соответствующих операндов. Существует также **постфиксная запись**. В ней оператор находится справа от обоих операндов. Арифметическое выражение (8.1) в постфиксной записи имеет вид

$$2\,3 * 4\,5 * +. \tag{8.2}$$

Посмотрим, как можно вычислить выражение (8.2). Создадим пустой стек и пройдёмся в цикле по элементам строки, содержащей постфиксную запись.

1. Если текущий элемент — число, то кладём его в стек.
2. Если текущий элемент — оператор, то вынимаем из стека два числа, применяем оператор к этим числам и кладём результат в стек.

После цикла в стеке будет содержаться число, равное результату математического выражения. При реализации данного алгоритма нужно следить за тем, чтобы для операции «$-$», например, операнды располагались в правильном порядке, так как эта операция чувствительна к порядку операндов.

Проиллюстрируем этот алгоритм на записи

$$1\,2 + 2\,4 + *. \tag{8.3}$$

Рис. 8.2

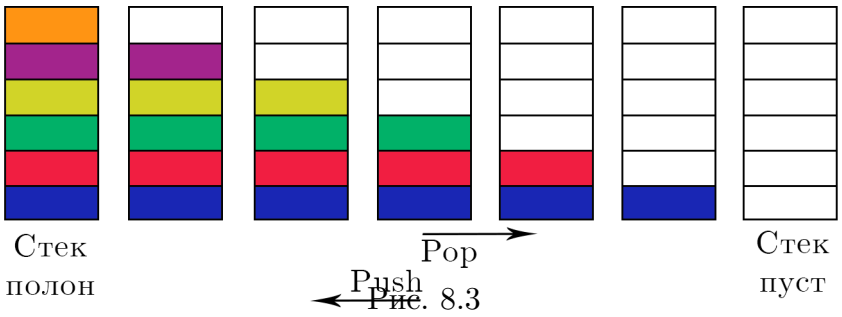


Рис. 8.3

Эта запись соответствует инфиксной записи

$$(1 + 2) * (3 + 4). \tag{8.4}$$

Заметим, что в постфиксной записи не нужны скобки для указания приоритета операций. Приоритет определяется последовательностью операций в строке.

Применим алгоритм вычисления постфиксной записи к (8.3):

1. Число 1: в стек.

   Стек: 1.

2. Число 2: в стек.

   Стек: 1 2.

3. Оператор +: достаём числа 1 и 2, считаем $1 + 2 = 3$ и кладём число 3 в стек.

   Стек: 3.

4. Число 3: в стек.

   Стек: 3 3.

5. Число 4: в стек.

   Стек: 3 3 4.

6. Оператор +: достаём числа 3 и 3, считаем $3 + 4 = 7$ и кладём число 7 в стек.

   Стек: 3 7.

7. Оператор $*$: достаём числа 3 и 7, считаем $3 * 7 = 21$ и кладём число 21 в стек.

   Стек: 21.

Итак, результат вычисления записи (8.3) равен 21.

Если возможны также и унарные операторы, например, унарный «минус», возвращающий для числа x число $-x$, то при обработке такого оператора нужно доставать из стека только одно число.

В языке C также есть один триарный оператор, то есть принимающий три аргумента. Это тернарный оператор «`?:`». Если верно условие, стоящее перед знаком «?», то выполняется выражение между знаками «?» и «:», иначе выполняется выражение, стоящее после знака «:». Например, $(a < b)?a = 7 : b = 3$. Соответственно, если в выбранной

Рис. 8.4

арифметике присутствуют триарные операторы, то при их обработке нужно вынимать из стека три числа.

Перейдём к реализации стека, хранящего целочисленные значения. Затем можно будет переделать реализацию для любого другого типа данных. Пока размер стека небольшой, можно реализовать его на основе массива. Создадим массив целых чисел из 10 элементов и обозначим его ячейки как $a_0, a_1, \dots, a_9$. Изначально в ячейках массива лежит какой-то мусор. Заведём целое число-счётчик *top*, задающее, насколько занят стек. Его можно определить двумя способами.

1. Можно сделать его равным либо номеру последней занятой ячейки. Тогда *top* будет принимать значения от 0 до 10 включительно. При пустом стеке $top = 0$, поскольку ячейка с номером «0» и является первой пустой.

2. Можно сделать его равным либо номеру первой незанятой ячейки. Тогда *top* будет принимать значения от -1 до 9 включительно. При пустом стеке $top = -1$.

Будем пользоваться первым вариантом, поскольку он удобнее для описания операций над стеком, а также потому что людям приятнее работать с неотрицательными числами (рис. **??**).

Имеет смысл написать структуру `Stack`, хранящую массив a и число *top*, потому что стеков в программе может быть много. [1]

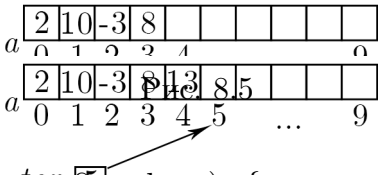
Нужно выделить память под стек. Создадим переменную *st* типа `Stack`. Как уже говорилось, в ней содержатся поля a и *top*. При создании этой структуры память под неё выделяется автоматически. Напишем процедуру `create`, чтобы выставить начальные значения массива. Далее каждое число, которое нужно положить в стек, считывается с клавиатуры. Затем содержимое стека распечатывается с помощью процедуры `print`. Для тестирования программы достаточно положить в стек числа $0, 1, 2, \dots, 9$. Тогда из стека должны быть извлечены числа $9, 8, \dots, 0$.

Чтобы распечатать массив, нужно вывести на экран только первые *top* элементов. В цикле пройдёмся по элементам массива и выведем их функцией `printf`:

```
void print(struct Stack s) {
        for(int i=0; i<s.top; i++)
                printf(%d\n, s.a[i]);
}
```

Теперь займёмся содержанием процедуры `create` (или `init`). Нулевые начальные значения можно не задавать, так как мусор, который был в ячейках до этого, просто затрётся новыми значениями, помещаемыми в ячейки при работе программы, а на экран он выведен не будет. *st* — это локальная переменная, по умолчанию она инициализируется мусором. Значит, для правильной работы программы переменной *top* нужно присвоить значение 0:

[1] Здесь должен быть листинг программы, который высвечивается проектором, но из-за особенностей




```
void create(struct Stack s) {
        s.top=0;
}
```

Однако такая процедура не изменяет состояния локальной переменной *st*, если её передать в качестве аргумента, потому что параметры функции передаются по значению. При вызове `create(st);` происходит побитовое копирование переменной *st* в новую переменную *s*. В ходе процедуры `create` изменяется *s*, а *st* остаётся прежним. Для того, чтобы действительно изменить содержание структуры *st*, можно воспользоваться одним из двух вариантов.

1. Можно написать в конце процедуры `return s;` вызов функции `create` теперь будет в виде `st=create(st);` Это неудобно, так как изменяется только переменная *top*, и производятся лишние операции копирования. При большом размере стека эти затраты существенно увеличивают время работы программы.

2. Можно в качестве аргумента передавать процедуре `create` не стек, а адрес стека. Тогда процедура выглядит так:

```
void create(struct Stack &s) {
        s->top=0;
}
```

Второй вариант, разумеется, предпочтительней. В процедуру `print` тоже лучше передавать адрес стека:

```
void print(struct Stack &s) {
        for(int i=0; i < s->top; i++)
                printf(%d\n, s->a[i]);
}
```

Во все остальные процедуры тоже будем передавать не значения, а указатели.

Пусть состояние стека описывается рисунком **??**. Нужно положить в стек число 13. Чтобы произвести операцию Push, нужно записать в ячейку с номером *top* это число и увеличить значение *top* на единицу:

```
void push(struct Stack *s, Data x) {
        s->a[s->top] = x;
        s->top ++;
}
```

Теперь состояние стека описывается рисунком **??**.

В процедуре Pop нужно присвоить новой переменной (назовём её *res*) последний элемент в стеке, уменьшить значение *top* на единицу и вернуть значение *res*:

съёмки видна только нижняя часть экрана, так что весь листинг привести не представляется возможным.

```
Data pop(struct Stack *s) {
        Data res = s->a[s->top-1];
        s->top --;
        return res;
}
```

Заметим, что все три строки можно записать всего одну:

```
Data pop(struct Stack *s) {
        return s->a[-- s->top];
}
```

Напомним, что запись $-x$ сначала уменьшает значение x на единицу, а затем с этим значением что-то производится. Значит, в предыдущем листинге сначала значение *top* уменьшается на единицу, затем возвращается элемент массива `s->a` с индексом, равным этому новому значению *top*.

Теперь, когда написаны все процедуры, можно запустить программу. Не забываем поправить вызовы процедур из функции `main()` — в них тоже нужно подставить адрес стека, а не сам стек. Программа работает так, как надо: сначала в стек кладутся значения от 0 до 9, затем они вынимаются в обратном порядке.

Добавим в функцию `main()` после заполнения стека в цикле следующую строчку:

```
push(st, 133);
```

Тогда программа попытается обратиться к памяти за пределами массива. Одно из возможных решений заключается в том, чтобы объявить размер массива через директиву `#DEFINE` и затем просто сделать его на единицу больше. Но это частичная мера: ведь может понадобиться положить в стек не один дополнительный элемент, а сотню, тысячу элементов или более.

Ещё одна возможная ошибка при работе с таким стеком: попытка вынуть элемент из пустого стека. Для решения этой проблемы можно дописать процедуру `is_empty(struct Stack *s)` и добавить условие `if(is_empty(&st))` при обращении к стеку:

```
int is_empty (struct Stack *s) {
        return s->top == 0;
}
...
int main()
{
        ...
        Data x;
        if(is_empty(&st)) x=pop(&st);
        ...
}
```

Выражение `s->top == 0` возвращает 1, если `s->top` равен нулю, и 0 в противном случае.

Хорошим решением будет заставить стек автоматически расширяться, если он полностью заполнен. Для этого нужно реализовать динамический захват памяти. Оставим пока начальное выделение памяти в процедуре `create` и модифицируем процедуру `push`, чтобы в ней происходило выделение дополнительной памяти при переполнении.

Пусть значение *top* совпадает с текущим размером массива. Тогда нужно сначала увеличить размер массива, и только потом помещать в массив новый элемент. [2]

```
s->a = realloc(s->a, sizeof(Data)*(s->top+1));
```

Теперь `s->a` будет указывать на адрес расширенной памяти. Старый адрес тоже лежал в указателе *a*.

Когда в стек кладётся первый элемент, *top* = 0. Чтобы `realloc` хорошо работал, нужно, чтобы первый адрес был равен либо какому-то фиктивному захвату, либо `null`. В последнем случае нужно использовать функцию `malloc`, и для этого и нужна функция `create`.

Можно также сделать стек автоматически сужающимся. Но в данной версии ограничимся только расширением. Добавим директиву `#include <stdlib.h>`, так как в этом заголовочном файле содержится функция `realloc`. В функцию `print` добавим строчку, выводящую результат процедуры `is_empty`. Скомпилируем, запустим программу и убедимся, что она работает. Пока в стеке что-то есть, на экран выводится `is_empty=0`, а после удаления последнего элемента `is_empty=1`.

Каждое выполнение функции `push` вызывает функцию `realloc`. Для экономии памяти можно было бы высвобождать освобождённую ячейку при выполнении процедуры `push`. Но `realloc` — «дорогая» по времени операция, и если бы количество вызовов процедур `push` было достаточно велико, то `realloc` бы заметно тормозил выполнение программы. Можно улучшить стек таким образом: выполнять расширение памяти не при каждом вызове функции `push`, а значительно реже. Добавим в структуру `Stack` переменную *capacity*, в которой будет храниться количество ячеек, выделенных для стека. В функции `create` можно задать начальное значение *capacity*, например, 10, а можно и оставить выбор начальной ёмкости за пользователем функции `create`:

```
void create(struct Stack *s, int capacity) {
        s->a = malloc(capacity*sizeof(Data));
        s->top = 0;
        s->capacity=capacity;
}
```

Теперь изменим функцию `push`. Пусть, например, *capacity* = *top*. Тогда при вызове `push` нужно захватывать новую память функцией `realloc`. Рассмотрим два варианта вызова функции `realloc`:

1. при каждом вызове realloc добавляется 100 ячеек;

2. при каждом вызове realloc добавляется 20% от существующего количества ячеек.

Если в программе используется много стеков малого размера, то выгодно использовать стратегию +100: если в каком-то стеке требуется больше места, то дополнительных 100 ячеек, скорее всего, будет достаточно. Если же нужны стеки большого размера, то более выгодна стратегия +20%, так как в этом случае потребуется меньше вызовов `realloc`.

```
void push(struct Stack *s, Data x) {
```

[2] Здесь должен быть листинг, но его не видно на той части экрана, которая есть на видео.

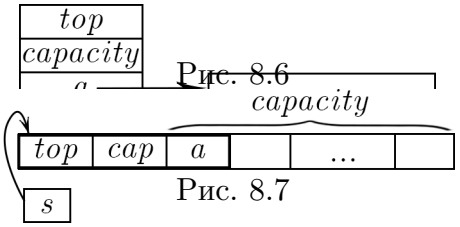


Рис. 8.6

Рис. 8.7

```
        if(s->top == s->capacity)
        {
                s->capacity += 100;
                s->a = realloc((s->capacity) * sizeof(Data));
        }
        s->a[s->top] = x;
        s->top ++;
}
```

Теперь переменные *top* и *capacity* разделены: размер выделенной памяти (в элементах) определяется переменной *capacity*, а количество занятых ячеек — переменной *top*.

Поскольку память динамически расширяется, то после выполнения программы её нужно динамически освобождать. Нужно написать функцию `destroy`, в которой это будет осуществляться. Чтобы обезопаситься от ошибки двух вызовов функции `destroy`, нужно сделать указатель `s->a` равным `null`. Функцию `destroy` нужно вызывать в конце функции `main`. Также можно написать функцию `clear`, которая очищает, но не удаляет стек.

```
void destroy(struct Stack *s) {
        free(s->a);
        s->a = null;
}
void clear(struct Stack *s) {
        s-> top=0;
}
```

Можно также ввести функцию `get`, которая будет возвращать верхний элемент в стеке, но не будет вынимать его из стека. Также можно добавить функцию `create2`, которая имеет только аргумент *capacity*, и эта функция сама возвращает переменную типа `struct Stack *`. Функция `create` принимает структуру *st*, созданную в функции `main`. Эту структуру можно изобразить на рис. **??**. Память, на которую указывает *a*, выделяется динамически. Если же воспользоваться функцией `create2`, то не нужно в функции main создавать переменную *st*. Достаточно только присвоить новой переменной *st* типа `struct Stack *` результат функции `create2`.

```
struct Stack * create2(int capacity) {
        struct Stack *s = malloc(sizeof(struct Stack));
        s->a = malloc(capacity*sizeof(Data));
        s->top = 0;
        s->capacity = capacity;
        return s;
}
```

Итак, на данной лекции стек был реализован на основе динамического массива. Массив данных должен в этом случае лежать в памяти после *top* и *capacity*, так как он должен быть расширяемым. Стек можно реализовать и на основе списков, о которых будет идти речь на следующем занятии. Также на основе списков будет реализована структура данных «очередь».