Курс «Алгоритмы и алгоритмические языки»

Лекция 10

Динамическое распределение памяти

- Функция

  `void *malloc(size_t size)`

  выделяет область памяти размером **`size`** байтов и возвращает указатель на выделенную область памяти.
  Если память не выделена (например, в системе не осталось свободной памяти требуемого размера), возвращаемый указатель имеет значение **`NULL`**.

- Поскольку результат операции **`sizeof`** имеет тип **`size_t`** и равен длине операнда в байтах, в качестве **`size`** можно использовать результат операции **`sizeof`**.

- Тривиальные примеры:

  (1) Выделение непрерывного участка памяти объемом 1000 байтов:

  ```
  char *p;
  p = (char *) malloc (1000);
  ```

  (2) Выделение памяти для 50 целых:

  ```
  int *p;
  p = (int *) malloc (50 * sizeof(int));
  ```

Динамическое распределение памяти

- Функция

  `void free(void *p)`

  возвращает системе выделенный ранее участок памяти с указателем **`p`**.

- ***Важное замечание***: Аргументом функции **`free()`** обязательно должен быть указатель **`p`** на участок памяти, выделенной ранее функцией **`malloc()`**.
  Вызов функции **`free()`** с неправильным указателем не определен и может привести кразрушению системы распределения памяти.
  Вызов функции **`free()`** с указателем **`NULL`** не приводит ни к каким действиям (C99).

- Функции **`malloc()`**и **`free()`**входят в состав библиотеки **`stdlib.h`**.

Динамическое распределение памяти

- ***Пример.*** Динамическое выделение памяти для строки:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main (void) {
  char *s;
  int t;

  s = (char *) malloc (80 * sizeof (char));
  if (!s) {
    printf ("требуемая память не выделена. \n");
    return 1; /* исключительная ситуация */
  }
  fgets (s, 80, stdin); s[strlen (s) - 1] = '\0';
  /* посимвольный вывод перевернутой строки на экран */
  for (t = strlen(s) - 1; t >= 0; t--)
    putchar (s[t]);
  free (s);
  return 0;
}
```

Динамическое распределение памяти

- ***Пример.*** Динамическое выделение памяти для двухмерного целочисленного массива (матрицы):

```
#include <stdio.h>
#include <stdlib.h>
int pwr (int a, int b) {
  int t = 1;
  for (; b; b--) t *= a;
  return t;
}

int main (void) {
  int *p[6];
  int i, j;
  for (i = 0; i < 6; i++)
    if (!(p[i] = (int *) malloc (4 * sizeof (int))))  {
      printf ("требуемая память не выделена. \n");
      exit(1);
    }

  for (i = 1; i < 7; i++)
    for (j = 1; j < 5; j++)
      p[i – 1][j – 1] = pwr (i, j);
  for (i = 1; i < 7; i++) {
    for (j = 1; j < 5; j++)
      printf ("%10d ", p[i – 1][j – 1]);
    printf ("\n");
  }
  /* Цикл с освобождением памяти */
  return 0;
}
```

Динамическое распределение памяти

- ***Пример.*** Динамическое выделение памяти для двухмерного целочисленного массива (матрицы):

```
#include <stdio.h>
#include <stdlib.h>

int pwr (int a, int b) {
  int t = 1;
  for (; b; b--)
    t *= a;
  return t;
}

int main (void) {
  int *p[6];
  int i, j;
  for (i = 0; i < 6; i++)
    if (!(p[i] = (int *) malloc (4 * sizeof (int))))  {
      printf ("требуемая память не выделена. \n");
      exit(1);
    }
```

Динамическое распределение памяти

- ***Пример.*** Динамическое выделение памяти для двухмерного целочисленного массива (матрицы):

```
  for (i = 1; i < 7; i++)
    for (j = 1; j < 5; j++)
      p[i – 1][j – 1] = pwr (i, j);
  for (i = 1; i < 7; i++) {
    for (j = 1; j < 5; j++)
      printf ("%10d ", p[i – 1][j – 1]);
    printf ("\n");
  }
  for (i = 0; i < 6; i++)
    free (p[i]);
  return 0;
}
```

Динамическое распределение памяти

- Состав функций динамического распределения памяти библиотеки **stdlib** (заголовочный файл **<stdlib.h>**)

```
void *malloc (size_t size);
void free (void *p);
void *realloc (void *p, size_t size);
void *calloc(size_t num, size_t size);
```

- Функция **void *realloc(void *p, size_t size)** согласно стандарту Си99 сначала выполняет **free (p)**, а потом **p = malloc (size)**, возвращая новое значение указателя **p**. При этом значения первых **size** байтов новой и старой областей совпадают.
- Функция **void *calloc(size_t num, size_t size)** работает аналогично функции **malloc(size1)**, где **size1** = **num * size** (т.е. выделяет память для размещения массива из **num** объектов размера **size**).

  Выделенная память инициализируется нулевыми значениями

Динамические структуры данных

- **Стек** (*stack*) – это динамическая последовательность *элементов*, количество которых изменяется, причем как добавление, так и удаление элементов возможно только с одной стороны последовательности (вершина стека).
- Стек можно организовать на базе массива **`stack[MAX]`**, где константа **`MAX`** задает максимальную глубину стека.
  Работа с таким стеком осуществляется с помощью функций:

  `push()` – *затолкать* элемент в стек;

  `pop()` – *вытолкнуть* элемент из стека;

  и глобальной переменной **`tos`** (*top of stack*).

Динамические структуры данных

- Организация стека на динамической памяти.

```
struct stack {
  int sp;   /* Текущая вершина стека */
  int sz;   /* Размер массива */
  char *stack;
} stack = { .sp = -1, .sz = 0, .stack = NULL };

static void push (char c) {
  if (stack.sz == stack.sp + 1) {
    stack.sz = 2*stack.sz + 1;
    stack.stack = (char *) realloc (stack.stack,
                  stack.sz*sizeof (char));
  }
  stack.stack[++stack.sp] = c;
}
```

Динамические структуры данных

- Организация стека на динамической памяти.

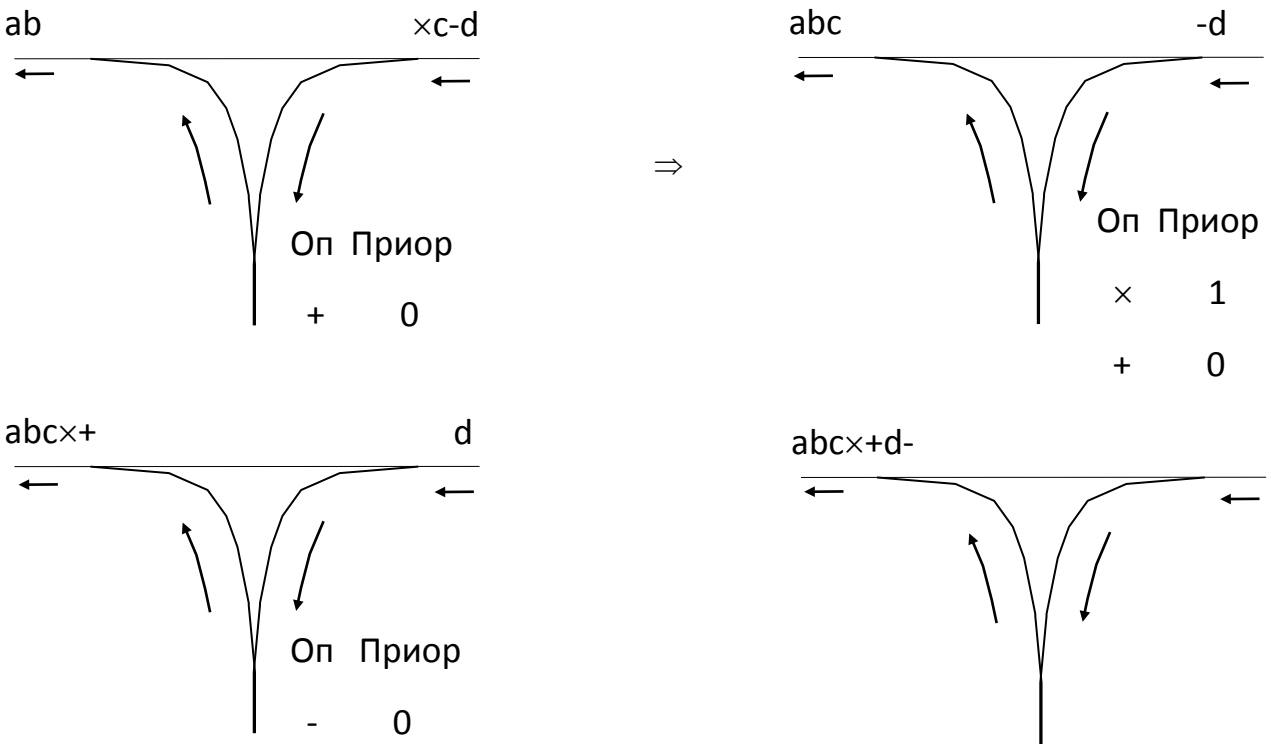
```
struct stack {
  int sp;   /* Текущая вершина стека */
  int sz;   /* Размер массива */
  char *stack;
} stack = { .sp = -1, .sz = 0, .stack = NULL };

static char pop (void) {
  if (stack.sp < 0) {
    fprintf (stderr, "Cannot pop: stack is empty\n");
    return 0;
  }
  return stack.stack[stack.sp--];
}

static int isempty (void) {
  return stack.sp == -1;
}
```

Динамические структуры данных

- **Пример.** Перевод арифметического выражения в обратную польскую запись (постфиксную).

a + b × c - d → abc × + d -

c × (a +b) - (d + e)/f → cab + × de + f/-



Динамические структуры данных

- **Пример.** Перевод арифметического выражения в обратную польскую запись.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

/* Считывание символа-операции или переменной */
static char getop (void) {
  int c;
  while ((c = getchar ()) != EOF && isblank (c))
    ;
  return c == EOF || c == '\n' ? 0 : c;
}
```

Динамические структуры данных

- **Пример.** Перевод арифметического выражения в обратную польскую запись.

```
/* Является ли символ операцией */
static int isop (char c) {
  return (c == '+') || (c == '-') || (c == '*')
         || (c == '/');
}
/* Каков приоритет символа-операции */
static int prio (char c) {
  if (c == '(')
    return 0;
  if (c == '+' || c == '-')
    return 1;
  if (c == '*' || c == '/')
    return 2;
  return -1;
}
```

Динамические структуры данных

- **Пример.** Перевод арифметического выражения в обратную польскую запись.

```
int main (void) {
  char c, op;

  while (c = getop ()) {
    /* Переменная-буква выводится сразу */
    if (isalpha (c))
      putchar (c);
    /* Скобка заносится в стек операций */
    else if (c == '(')
      push (c);
    else <...>
```

Динамические структуры данных

- **Пример.** Перевод арифметического выражения в обратную польскую запись.

```
/* Операция заносится в стек в зависимости от приоритета */
else if (isop (c)) {
      while (! isempty ()) {
        op = pop ();
        /* Заносим, если больший приоритет */
        if (prio (c) > prio (op)) {
          push (op); break;
        } else
        /* Иначе выталкиваем операцию из стека */
          putchar (op);
      }
      push (c);
    } else <...>
```

Динамические структуры данных

- **Пример.** Перевод арифметического выражения в обратную польскую запись.

```
/* Скобка выталкивает операции до парной скобки */
} else if (c == ')')
      while ((op = pop ()) != '(')
        putchar (op);
  }
/* Вывод остатка операций из стека */
  while (! isempty ())
    putchar (pop ());
  putchar ('\n');
  return 0;
}
```