Лекция 10 Динамические структуры данных

10.1. Динамическое распределение памяти.

10.1.1. Выделение памяти с помощью функции **`malloc()`**. Память выделяется во время выполнения программы.

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`**.

Замечание. Как уже отмечалось, тип **`size_t`**, который определяется как беззнаковый целочисленный тип результатов операции **`sizeof`**. Поскольку **`sizeof`** выдает длину типа в байтах, в качестве **`size`** можно использовать результат операции **`sizeof`** (Раздел 7.17 Стандарта Си99).

Тривиальные примеры:

(1) Выделение непрерывного участка памяти объемом 1000 байтов:

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

(2) Выделение памяти для 50 целых:

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

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 (в частности, с указателем **`null`**) приводит к разрушению системы распределения памяти.

10.1.2. ***Пример***. Динамическое выделение памяти для массива байтов (строки):

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

     s = malloc(80);

     if (!s) {
        printf ("требуемая память не выделена. \n");
        return 1 /* исключительная ситуация */
     }
     gets(s); /* ввод строки */
     /* посимвольный вывод строки на экран */
     for(t = strlen(s) - 1; t>=0; t--) putchar(s[tmp]);
     free(s);
     return 0;
}
```

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

```
#include <stdio.h>
#include <stdlib.h>
int pwr(int a, int b)
int main() {
     int (*p)[6];   /* (*p), т. к. у []приоритет выше,
                     * чем * у *
     register int i, j;
     p = malloc(24 * sizeof(int));
     if (!s) {
          printf ("требуемая память не выделена. \n");
          exit(1)
     }
     for (j = 1; j < 7; j++)
         for (i = 1; i < 5; i++)
              p[i - 1][j - 1] = pwr(j, i);

    for (j = 1; j < 7; j++) {
         for (i = 1; i < 5; i++)
              printf("%10d "; p[i - 1][j - 1]);
         printf("\n");
    }
    return(0);
}
int pwr(int a, int b){
     int t = 1;
     for (; b; b--) t *= a ;
     return t;
}
```

10.2. Другие функции динамического распределения памяти из библиотеки **stdlib**.

10.2.1. Помимо функции **void *malloc(size_t size)** и функции **void free(void *p)** библиотека **stdlib** (заголовочный файл **<stdlib.h>**) содержит еще две функции динамического распределения памяти: функцию **void *realloc(void *p, size_t size)** и функцию **void *calloc(size_t** *num*, **size_t size)**.

10.2.2. Функция **void *realloc(void *p, size_t** *size***)** согласно стандарту Си99 сначала выполняет **free(*p)**, а потом **malloc(size)**, возвращая новое значение указателя. При этом значения первых **size** байтов новой и старой областей совпадают.

10.2.3. Функция **void *calloc(size_t** *num*, **size_t** *size***)** работает аналогично функции **malloc(***size1***)**, где *size1 = num * size* (т.е. выделяет память, достаточную для размещения массива, содержащего *num* объектов размера *size*).

10.3. Динамические структуры данных.

10.3.1. ***Очередь***.

10.3.1.1. Очередь (queue) – это линейный список информации, работа с которой происходит по принципу FIFO. Для списка можно использовать статический массив (количество элементов = наибольшей допустимой длине очереди MAX).

10.3.1.2. Работа с очередью осуществляется с помощью двух функций:

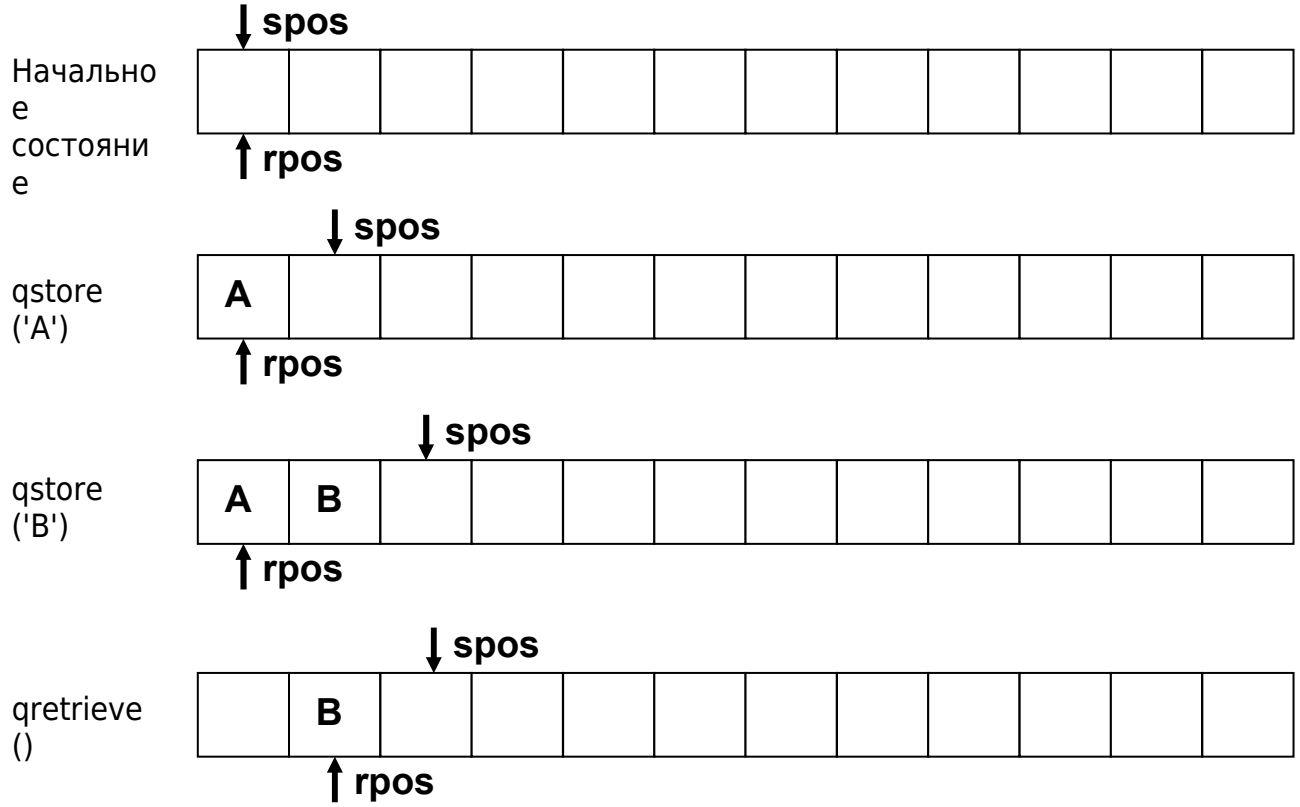
`qstore()` – *поместить* элемент в конец очереди;

`qretrieve()` – *удалить* элемент из начала очереди;

и двух глобальных переменных:

`spos` (индекс первого свободного элемента очереди: его значение <= **`MAX`**) и **`rpos`** (индекс очередного элемента, подлежащего удалению: «кто первый?»)

10.3.1.3. Тексты функций п°.10.3.1.2.




```
#define MAX    67

int *queue[MAX];
int spos = 0, rpos = 0;

void qstore(int *q) {
   if(spos == MAX) {
      printf("Очередь переполнена \n");
      return;
   }
   queue[spos] = q;
   spos++;
```

```
}

int *qretrieve() {
  if(rpos == spos) {
     printf("Очередь пуста \n");
     return '\0';
  }
  rpos++;
  return queue[rpos - 1];
}
```

10.3.1.4. Улучшение – «циклическая» очередь. Чтобы не терять начало массива,

```
#define MAX    67

int *queue[MAX];
int spose = 0, rpose = 0;

void qstore(int *q) {
  if(spos + 1 == rpos || (spos + 1 == MAX & !rpos) {
     printf("Очередь переполнена \n");
     return;
  }
  queue[spos] = q;
  spos++;
  if(spos == MAX) spos = 0;
}

int *qretrieve() {
  if(rpos == MAX) rpos = 0;
  if(rpos == spos) {
     printf("Очередь пуста \n");
     return '\0';
  }
  rpos++;
  return queue[rpos - 1];
}
```

Зацикленная очередь переполняется, когда **spos** находится непосредственно перед **rpos**, так как в этом случае запись приведет к **rpos == spos**, т.е. к пустой очереди.

10.3.2. ***Стек***.

10.3.2.1. Стек (*stack*) – это линейный список информации, работа с которой происходит по принципу *LIFO*.

10.3.2.2. Работа со стеком, организованном на базе массива **stack[MAX]**, осуществляется с помощью двух функций:

push() – *затолкать* элемент в стек;

pop() – *вытолкнуть* элемент из стека;

и глобальной переменной **tos** (*top of stack*). По сравнению с п°.10.3.1. почти ничего нового.

10.3.2.3 Организация стека на базе динамической памяти. Помимо указателя **tos** потребуется еще один указатель (*bottom of stack*). В случае массива длины **MAX** роль **bos** играет конец массива.

10.3.2.4. Тексты функций

```
int *st; // указатель стека
int *tos;
int *bos;

void push (int i) {
     if(st > bos) {
          printf("Переполнение стека\n");
          return;
     }
     *st = i;
     st=++;
}

int pop(void) {
     st--;
     if(st < tos) {
          printf("Стек пуст\n");
          return 0;
     }
     return *st;
}
```

Перед использованием этих функций стек необходимо *инициализировать*, т.е. выполнить операторы:

```
st = malloc (m); tos = st; bos = st + m;
```

10.3.2.5. В случае переполнения стека можно не только известить об этом пользователя, но и увеличить стек с помощью операторов:

```
*realloc(st, m + n); bos += n;
```

10.3.3. ***Применение стека***: перевод арифметического выражения в обратную польскую запись.