

Основы программирования

Лабораторная работа 7

Тема: Динамическое распределение памяти

Цель лабораторной работы

Целями данной работы являются получение навыков использования указателей для работы с динамическим распределением памяти.

Краткие теоретические сведения

Динамическое распределение памяти

В дополнение к *глобальным* и *локальным* объектам в C++ можно создавать **динамические объекты**. Продолжительность их жизни не зависит от того, где они созданы. *Динамические объекты* существуют, пока не будут удалены явным образом. Они размещаются в **динамически распределяемой области памяти** (*heap*, *куча*), и для их создания применяется *динамическое распределение памяти*. Это означает, что программа выделяет память для данных во время своего выполнения.

Функции динамического распределения памяти языка C

Распределение памяти с помощью malloc()

Функция **malloc()** возвращает адрес на первый байт некоторой области памяти определенного размера в байтах, которая была выделена из кучи. Прототип:

```
void *malloc(size_t количество_байтов);
```

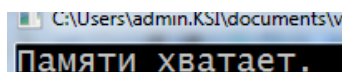
Здесь *количество_байтов* – размер памяти, необходимой для размещения данных.

При успешном выполнении *malloc()* возвращает указатель на первый байт выделенного непрерывного участка памяти.

```
char *p;  
p = (char*) malloc(1); /* выделение 1 байта */  
*p = 'a';
```

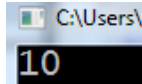
Поскольку динамически распределяемая область памяти не бесконечна, то если в динамически распределяемой области памяти недостаточно свободной памяти для выполнения запроса, то *malloc()* возвращает NULL. Поэтому при каждом размещении данных необходимо проверять, состоялось ли оно:

```
int *p;  
p = (int*) malloc(4);  
if(!p) {  
    printf("Нехватка памяти.\n");  
    exit(1);  
}  
printf("Памяти хватает.\n");
```



Для повышения мобильности (переносимости программы) используется оператор **sizeof()**, который возвращает размер *типа* или *переменной* заданного типа:

```
int *p;
p = (int*) malloc(sizeof(int));
if(!p) {
    *p = 10;
    printf("%d", *p);
}
```



Освобождение памяти с помощью free()

Функция **free()** возвращает системе (освобождает) участок памяти, выделенный ранее с помощью функции *malloc()*. Прототип:

```
void free(void *p);
```

Здесь *p* – указатель на участок памяти, выделенный перед этим функцией *malloc()*.

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

Функция *free()* может принимать **нулевой указатель**. В этом случае она ничего не делает.

После освобождения памяти адрес в указателе не переписывается, так что он остается ссылаться на адрес участка памяти, который помечен как *свободный*. Однако значение по данному адресу может быть уже другим.

```
int *p = (int*) malloc(sizeof(int));
*p = 10;
printf("Адрес: %p, значение: %d\n", p, *p);
free(p); // освобождение
printf("Адрес: %p, значение: %d\n", p, *p);
```

A screenshot of a console window showing two lines of output. The first line is 'Адрес: 006D1320, значение: 10' and the second line is 'Адрес: 006D1320, значение: -17891602'. The text is white on a black background.

```
Адрес: 006D1320, значение: 10
Адрес: 006D1320, значение: -17891602
```

Поэтому рекомендуется после освобождения **занулять** указатель:

```
int *p = (int*) malloc(sizeof(int));
free(p); // освобождение
p = nullptr;
```

Динамическое распределение памяти языка C++

Распределение памяти с помощью new

Оператор **new** выделяет место в динамической памяти для объекта и возвращает указатель на этот объект. Синтаксис:

```
тип_данных* указатель = new тип_данных;
```

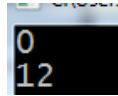
Пример создания динамического объекта:

```
int *ptr = new int;
```

Оператор *new* создает новый объект типа *int* в динамической памяти и возвращает указатель на него. Значение такого объекта неопределенно.

Пример:

```
int *p1 = new int(); // значение по умолчанию для int = 0
cout << *p1 << endl;
int *p2 = new int(12);
cout << *p2 << endl;
```



```
0
12
```

Освобождение памяти с помощью delete

Оператор **delete** получает указатель на динамический объект и удаляет его из памяти:

```
delete указатель;
```

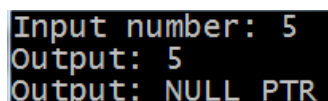
Пример:

```
int *p = new int;
delete p; // удаление объекта
```

Применять данный оператор к **нулевому указателю** безопасно.

Однако результат повторного применения операции **delete** к одному и тому же *ненулевому* указателю не определен. Поэтому, как и в случае с функцией *free()*, если вам указатель не нужен будет в дальнейшем лучше его **обнулить**. Например:

```
int* createPtr(int value)
{
    int *ptr = new int(value); // здесь создается
    return ptr;
}
void usePtr(int *&ptr)
{
    if(ptr == nullptr) // проверка
    {
        cout << "Output: NULL PTR" << endl;
        return;
    }
    cout << "Output: " << *ptr << endl; // здесь используется
    delete ptr; // здесь освобождается
    ptr = nullptr; // здесь зануляется
}
int main()
{
    int a;
    cout << "Input number: ";
    cin >> a;
    int *p = createPtr(a);
    usePtr(p); // вызываем первый раз
    usePtr(p); // вызываем второй раз
    return 0;
}
```



```
Input number: 5
Output: 5
Output: NULL PTR
```

В функции *usePtr()* получаем ссылку на указатель на динамический объект, выделенный функцией *createPtr()*. Однако после выполнения функции *usePtr()* этот объект автоматически не удаляется из памяти. Поэтому его надо явным образом удалить с помощью оператора *delete*. При этом также необходимо **занулить** указатель, чтобы избежать обращения к освобожденной памяти.

Динамические массивы

Кроме отдельных динамических объектов в языке C/C++ мы можем использовать **динамические массивы**. Синтаксис:

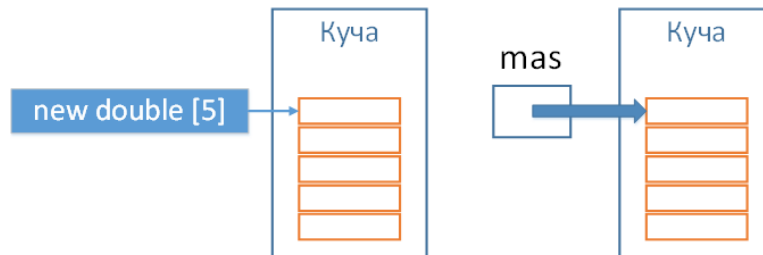
```
указатель = (тип*) malloc (размер_массива * sizeof(тип));  
указатель = new тип [размер_массива];
```

Размер массива – количество переменных указанного типа, под которое выделяется память.

Необходим для определения размера выделяемой области памяти.

Пример:

```
// динамические массивы из 5 чисел  
int *arr = (int*) malloc( 5 * sizeof(int) );  
int *mas = new int[5];
```



Основной особенностью динамического массива является то, что его размер (объем выделенной области) может определяться не заранее в коде программы, а вычисляться **во время выполнения** программы.

```
srand(time(nullptr));  
int n; // размер массива  
cout << "Count of elements: ";  
cin >> n; // определяем размер  
if(n > 0)  
{  
    int *nums = new int[n]; // создаем массив  
    for(int i=0; i<n; i++)  
    {  
        nums[i] = rand() % 100;  
        cout << nums[i] << " ";  
    }  
}  
else cout << "Wrong count";  
cout << endl;
```

C:\Users\admin.K31\documents\visual studio 2010\Project
Count of elements: 10
52 18 1 94 89 56 3 52 7 94

Освобождение памяти для динамического массива

Для удаления динамического массива и освобождения его памяти в языке C используется та же функция **free()**. Однако в языке C++ применяется специальная форма оператора **delete**:

```
delete [] указатель_на_динамический_массив;
```

Например:

```
int *p = new int[5]{ 1, 2, 3, 4, 5 };  
delete [] p; // удаление динамического массива
```

Поразрядные операции

Побитовые (поразрядные) операции выполняются над отдельными *разрядами* или *битами* значений переменных. Такие операции могут производиться только над **целыми числами**.

Оператор	Операция
Логические операции	
&	И
	ИЛИ
^	исключающее ИЛИ
~	НЕ (отрицание, дополнение к 1)
Операции сдвига	
>>	Сдвиг вправо
<<	Сдвиг влево

Рассмотрим данные операции по отдельности.

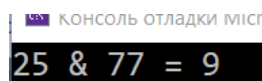
Унарная операция «~» преобразовывает все *единицы* в *нули* и все *нули* в *единицы* в двоичной записи числа. Данную операцию называют также операцией «*дополнение*».

Поразрядная операция И обозначается символом «&». Двоичная **операция &** создает новое значение за счет выполнения поразрядного сравнения двух операндов. Для каждой позиции результирующий разряд будет иметь значение 1 (*true*) только в случае, если соответствующие разряды обоих операндов имеют значение 1. Например:

v1	0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 (25)
v2	0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 1 (77)
v1 & v2	0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 (9)

В результате поразрядной операции «&» над двумя числами 25 и 77, которые, например, представляют тип *short* (16 бит), получили новое число – 9:

```
short v1 = 25;
short v2 = 77;
short res = v1 & v2;
cout << v1 << " & " << v2 << " = " << res;
```



25 & 77 = 9

Поразрядная операция & называется также *поразрядной конъюнкцией*, или *побитовым логическим умножением*.

Поразрядная операция ИЛИ обозначается символом «|». Когда над двумя значениями производится операция поразрядно ИЛИ, то последовательно сравниваются значения всех битов при двоичном представлении этих значений. Если при этом соответствующий бит имеет значение 1 в первом или втором операнде, то результирующее значение будет равно 1. Например:

v1	0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 (25)
v2	0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 1 (77)
v1 v2	0 0 0 0 0 0 0 0 0 0 1 0 1 1 1 0 1 (93)

```
short v1 = 25;
short v2 = 77;
short res = v1 | v2;
cout << v1 << " | " << v2 << " = " << res;
```

КОНСОЛЬ ОТЛАДКИ msvc
25 | 77 = 93

Поразрядную операцию ИЛИ (|) обычно используют для установки заданных битов слова в 1. Ее также называют *включающей дизъюнкцией* или *побитовым логическим сложением*.

Операция исключающего ИЛИ обозначается символом «^». Результат данной операции равен ИСТИНА, если только один из операндов равен 1, иначе результат будет равен ЛОЖЬ.

v1	0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 (25)
v2	0 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 1 (77)
v1 ^ v2	0 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 0 (84)

```
int res = 25 ^ 77;
cout << "25 ^ 77 = " << res;
```

КОНСОЛЬ ОТЛАДКИ msvc
25 ^ 77 = 84

Поразрядная операция *исключающего ИЛИ* называется также *исключающей дизъюнкцией*.

Поразрядные операции сдвига

Оператор сдвига влево: <<. Когда оператор *сдвига влево* (<<) выполняется над некоторым значением, все биты, составляющие это значение, сдвигаются **влево**. Связанное с этим оператором число показывает количество бит, на которое значение должно переместиться. Биты, которые сдвигаются со старшего разряда, считаются *потерянными*, а на место младших битов всегда помещаются *нули*.

Оператор сдвига вправо: >>. Операция *сдвига вправо* (>>) сдвигает разряды левого операнда вправо на количество **позиций**, указываемое правым операндом. Выходящие за правую границу разряды *теряются*. Для типов данных без знака (*unsigned*) освобождаемые слева позиции заполняются *нулями*. Для знаковых типов данных результат зависит от используемой системы.

Пример:

unsigned char x	x после операции	значение x
x = 7	0000 0111	7
x = x << 1	0000 1110	14
x = x << 3	0111 0000	112
x = x << 2	1100 0000	192
x = x >> 1	0110 0000	96

Каждый сдвиг влево умножает на 2. Каждый сдвиг вправо делит на 2. Потеря информации произошла после операции x<<2 в результате сдвига за левую границу. Сдвиг вправо потерянную информацию не восстановил.

Задание

Разработать программу для каждой задачи из **общего** и **индивидуального** задания.

Во всех программах необходимо использовать **динамические** массивы. Размер массива должен вводить пользователь.

Все изменения массивов и строк в функциях **должны сохраняться**. Если сказано, что значение сохраняется в *переменной-аргументе*, то это значит, что все изменения с этой переменной **должны отразиться на оригинальной переменной**, переданной в функцию, а сама переменная-аргумент является **указателем**.

Все пользовательские функции должны **только выполнять вычисление и возвращать результат** в главную функцию (если в задаче на сказано иного). Все результаты работы программ необходимо выводить на экран в функции *main()*.

Методические указания

Выполните **общие** и **индивидуальные** задания.

При выполнении **общих** заданий не требуется отчет, но все еще нужна защита с объяснением кода программы.

При выполнении **индивидуальных** заданий необходимо выполнить следующие этапы **для каждой задачи отдельно**:

- 1) словесная постановка задачи;
- 2) анализ задачи и формальная постановка задачи;
- 3) проектирование (разработка алгоритма) – лучше всего в графической форме (блок-схема), но можно и в словесной форме или псевдокод;
- 4) реализация (кодирование, отладка);
- 5) тестирование.

Результаты выполнения индивидуальных заданий оформить в виде отчета.

Общие задания

Задание 1

Напишите функцию, которая получает в качестве аргумента **размер массива**. В функции нужно выделить память под целочисленный массив указанного размера, затем заполнить его *случайными числами*. После этого отсортируйте массив по возрастанию (метод сортировки пузырьком). Далее функция должна возвращать **указатель** на созданный массив.

В основной функции *main()* принимайте от пользователя введенный *размер* массива. Далее вызывайте написанную функцию и передавайте ей этот *размер*. Принимайте в качестве результата функции *указатель на массив* и выводите его на экран.

Размер массива: 10

Созданный массив:

6 9 13 22 23 40 55 78 81 89

Задание 2

Напишите программу, которая использует для хранения значений 8-ми флагов переменную размером в 1 байт (*char*). Изначально все флаги должны быть выключены (равны 0). Считывайте с клавиатуры числа от 1 до 8, которые говорят, какой из флагов переключить (если он равен 0, то сделать 1, иначе сделать 0). После каждого такого изменения выводить текущее состояние флагов в виде перечислений значений битов (начальное состояние – 00000000).

Для решения данной задачи можно использовать побитовый **оператор «&»**, а также **сдвиг влево**, чтобы в цикле перемещать 1 для дальнейшего использования.

```
Состояние флагов: 00000000
Переключить бит: 3
Состояние флагов: 00100000
Переключить бит: 5
Состояние флагов: 00101000
Переключить бит: 3
Состояние флагов: 00001000
```

Варианты индивидуальных заданий

Вариант 1

Написать функцию, которая принимает *строку*. Создает новую строку, в которую записываются все символы принятой строки, но при этом они дублируются («a» - «aa»). Возвращает полученную строку.

```
Введите строку:
Hello World
Результат:
HNeellllo  WWoorrlldd
```

Вариант 2

Написать функцию, которая принимает *строку* и число N. Создает новую строку из принятой строки, продублированной N раз. Возвращает эту строку.

```
Введите строку: Hello
Введите число: 3
Результат:
HelloHelloHello
```

Вариант 3

Написать функцию, которая принимает *строку* и *переменную-указатель*. Делит строку на две строки. Первая – первая половина исходной строки, и она запоминается в *переменной-аргументе*. Вторая – *вторая половина*, и она возвращается функцией в качестве результата.

```
Введите строку:
Hello World
Результат:
1 - Hello
2 - World
```


Вариант 4

Написать функцию-процедуру, которая принимает *строку*. Находит в строке символы, которые встречаются только **один раз**, и сохраняет их в отдельной строке. Возвращает эту строку.

```
Введите строку:
Hello World
Результат:
He Wrld
```

Вариант 5

Написать функцию, которая принимает *две строки*, создает *третью строку* из объединения **первой половины первой строки** и **второй половины второй строки**. Возвращает эту строку.

```
Введите строку 1: Hello World
Введите строку 2: Goodbye Sunny
Результат:
Hello Sunny
```

Вариант 6

Написать функцию, которая принимает *две строки*. Объединяет *две строки* в третью: вначале идет **короткая** строка, потом более **длинная**. Возвращает эту строку.

```
Введите строку 1: Hello World
Введите строку 2: Hi User!
Результат:
Hi User!Hello World
```

Вариант 7

Написать функцию, которая принимает *строку* и *переменную-символ*. Находит в строке наиболее часто встречаемый символ и сохраняет его в *переменной-аргументе*. Возвращает копию исходной строки, но без этого символа.

```
Введите строку: Hello World
Результат:
Символ - l
Строка - Neo Word
```

Вариант 8

Написать функцию, которая принимает *массив любого базового типа* через указатель **void *** и его тип (можно для этого использовать *typedef* или просто числовую переменную для обозначения номера типа). Копирует массив и возвращает его **копию**.

Вариант 9

Написать функцию-процедуру, которая принимает *строку*, *символ* и *переменную*. Создает копию исходной строки и удаляет из нее этот символ. Сохраняет **количество** удаленных символов в *переменной-аргументе*, а получившуюся строку возвращает в качестве результата.

```
Введите строку: Hello World
Символ: l
Результат:
Количество символов в строке = 3
Строка без символа: Neo Word
```

Вариант 10

Написать функцию, которая принимает *строку*, *числовую переменную N* и *символ*. Заменяет в строке каждый N-ый символ на *символ-аргумент*, а все изначальные *заменяемые* символы сохраняет в **отдельном массиве**. После этого возвращает *полученный* массив в качестве результата.

```
Введите строку: Hello World
Символ: !
Номер символа: 3
Результат:
Измененная строка: He!lo!Wo!ld
Замененные символы: l r
```

Вариант 11

Написать функцию, которая принимает *строку* и *переменную*. Делит строку на *две строки*. В первой строке все **четные** символы, и адрес этой строки запоминается в *переменной-аргументе*. Во второй строке все **нечетные** символы, и она возвращается функцией в качестве результата.

Вариант 12

Написать функцию, которая принимает *строку*. Создает копию исходной строки. Удаляет из нее все повторяющиеся подряд символы («aaa oo» - «a o») и возвращает ее в качестве результата.

```
Введите строку: Heeeello Woooorldd
Результат:
Helo World
```

Вариант 13

Написать функцию, которая принимает *строку* и *переменную*. Возвращает копию строки, где меняет местами соседние **четные** и **нечетные** символы. Количество символов, которые после замены не изменились (поменялись одинаковые) сохраняет в *переменной-аргументе*.

Вариант 14

Написать функцию, которая принимает *строку* и *переменную*. Возвращает копию строки без знаков препинания «!.,?»», а также в *переменной-аргументе* количество удаленных знаков.

```
Введите строку: Hello!!! World.
Результат:
Удалено знаков - 4.
Измененная строка: Hello World
```