ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

УПРАВЛЕНИЕ ДИСТАНЦИОННОГО ОБУЧЕНИЯ И ПОВЫШЕНИЯ КВАЛИФИКАЦИИ

Кафедра «Программное обеспечение вычислительной техники и автоматизированных систем»

# Учебно-методическое пособие

по дисциплинам

# «Объектно-ориентированное программирование», «Программирование»

Автор
Габрельян Б.В.

Ростов-на-Дону, 2018

# Аннотация

Учебно-методическое пособие предназначено для студентов очной формы обучения направлениям 02.03.03 Математическое обеспечение и администрирование информационных систем, 09.03.04 Программная инженерия

# Авторы

доцент, к.ф.-м.н.,
зав.каф. ПОВТиАС
Габрельян Б.В.

# Оглавление

**1. Лабораторная работа №1: Среда VISUAL STUDIO. Создание проекта, отладка, добавление библиотек. Стандартный ввод/вывод ........6**

1.1. Теория ........ 6
1.2. Задание к лабораторной работе ........14
1.3. Контрольные вопросы........15
1.4. Литература ........15

**2. Лабораторная работа №2: Операции и выражения в C/c++ ........15**

2.1. Теория ........15
2.2. Задание к лабораторной работе ........25
2.3. Контрольные вопросы........26
2.4. Литература ........27

**3. Лабораторная работа №3: Операторы C/C++........27**

3.1. Теория ........27
3.2. Задание к лабораторной работе ........38
3.3. Контрольные вопросы........39
3.4. Литература ........41

**4. Лабораторная работа №4: Работа с массивами ........41**

4.1. Теория ........41
4.2. Задание к лабораторной работе ........43
4.3. Контрольные вопросы........44
4.4. Литература ........44

**5. Лабораторная работа №5: Указатели в C/C++ ........44**

5.1. Теория ........44
5.2. Задание к лабораторной работе ........55
5.3. Контрольные вопросы........57
5.4. Литература ........58

**6. Лабораторная работа №6: Динамические структуры данных ........58**

6.1. Теория ........58
6.2. Задание к лабораторной работе ........63
6.3. Контрольные вопросы........64
6.4. Литература ........64

**7. Лабораторная работа №7: Файловый ввод и вывод..65**

7.1. Теория....................................................65
7.2. Задание к лабораторной работе ........................73
7.3. Контрольные вопросы......................................74
7.4. Литература.................................................75

**8. Лабораторная работа №8: Конструкторы и деструктор ...........................................................................75**

8.1. Теория....................................................75
8.2. Задание к лабораторной работе ........................78
8.3. Контрольные вопросы......................................79
8.4. Литература.................................................79

**9. Лабораторная работа №9: ССылки. Копирование и присваивание ........................................................80**

9.1. Теория....................................................80
9.2. Задание к лабораторной работе ........................82
9.3. Контрольные вопросы......................................83
9.4. Литература.................................................83

**10. Лабораторная работа №10: Динамические структуры данных на примере двусвязного списка.........84**

10.1. Теория....................................................84
10.2. Задание к лабораторной работе ........................85
10.3. Контрольные вопросы......................................86
10.4. Литература.................................................86

**11. Лабораторная работа №11: Перегрузка операций.86**

11.1. Теория....................................................86
11.2. Задание к лабораторной работе ........................90
11.3. Контрольные вопросы......................................90
11.4. Литература.................................................91

**12. Лабораторная работа №12: Статические поля и методы класса ........................................................91**

12.1. Теория....................................................91
12.2. Задание к лабораторной работе ........................93
12.3. Контрольные вопросы......................................94
12.4. Литература.................................................94

**13. Лабораторная работа №13: Наследование в C++ ..94**

13.1. Теория....................................................94
13.2. Задание к лабораторной работе ........................97

13.3. Контрольные вопросы....................................99
13.4. Литература ..............................................99
**14. Лабораторная работа №14: Обработка исключений . ............................................................. 100**
14.1. Теория ...................................................100
14.2. Задание к лабораторной работе .....................102
14.3. Контрольные вопросы..................................103
14.4. Литература ..............................................103

# 1. ЛАБОРАТОРНАЯ РАБОТА №1: СРЕДА VISUAL STUDIO. СОЗДАНИЕ ПРОЕКТА, ОТЛАДКА, ДОБАВЛЕНИЕ БИБЛИОТЕК. СТАНДАРТНЫЙ ВВОД/ВЫВОД

## 1.1. Теория

**I. Создание проекта**

1) Создать проект консольного приложения.

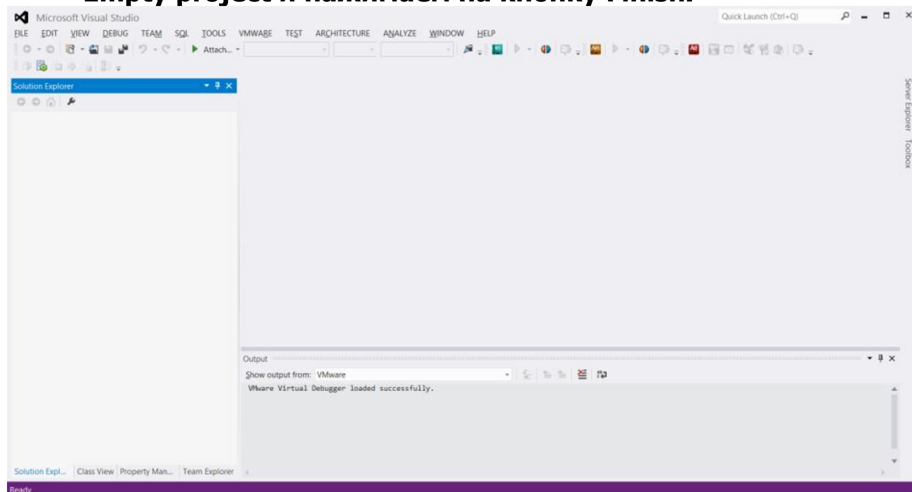
File-New-Project..., затем в правой части выбираем Win32 Console Application.

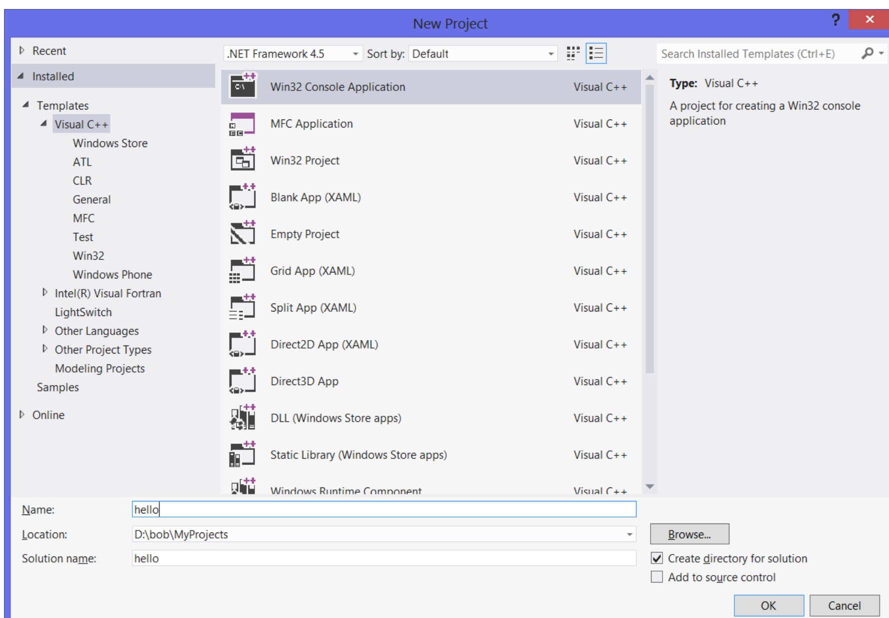
Заполняем поле Name, например, Name: hello

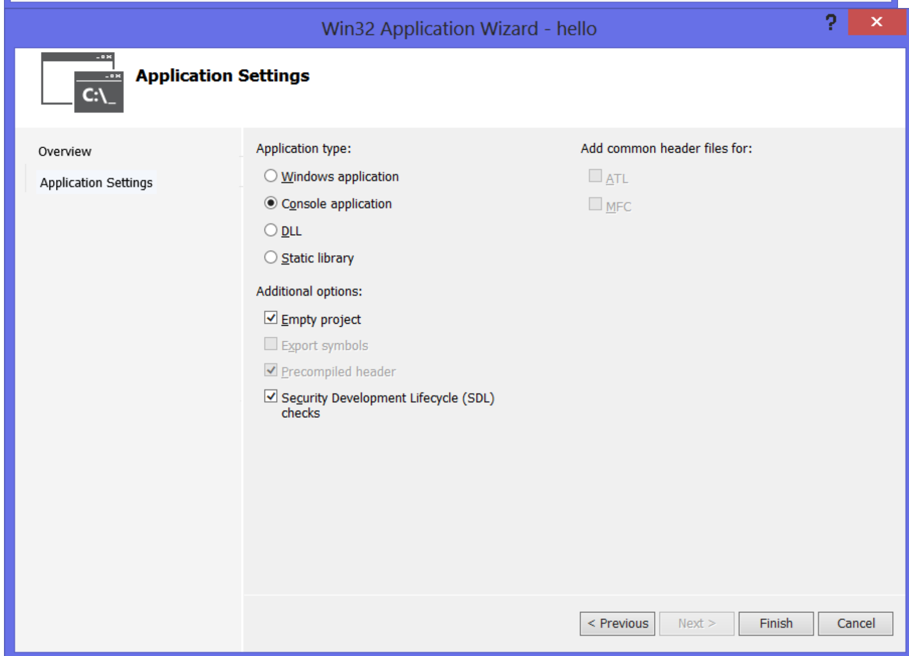
Запоминаем или задаем место расположения всех файлов нашего проекта в поле Location:

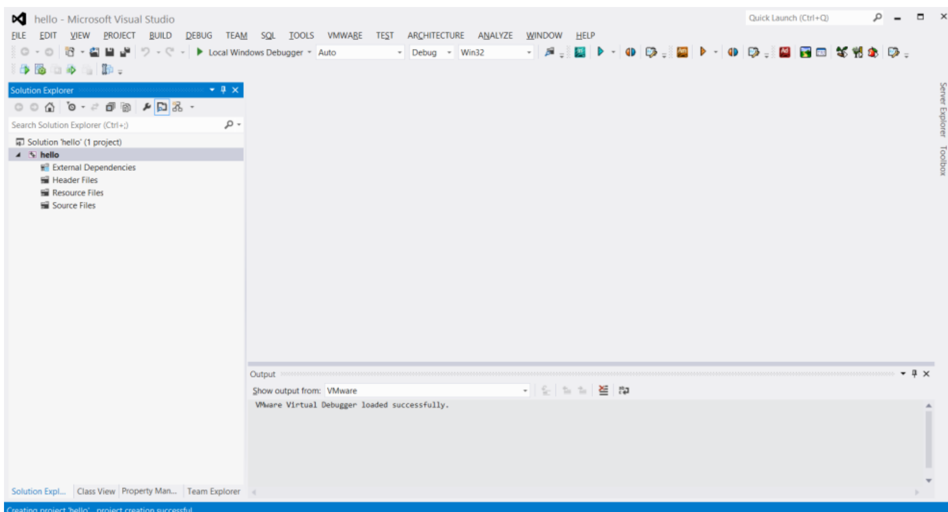
Появляется окно Welcome to the Win32 Application Wizard. **Нажимаем Next, затем отмечаем галочкой**

**Empty project и нажимаем на кнопку Finish.**

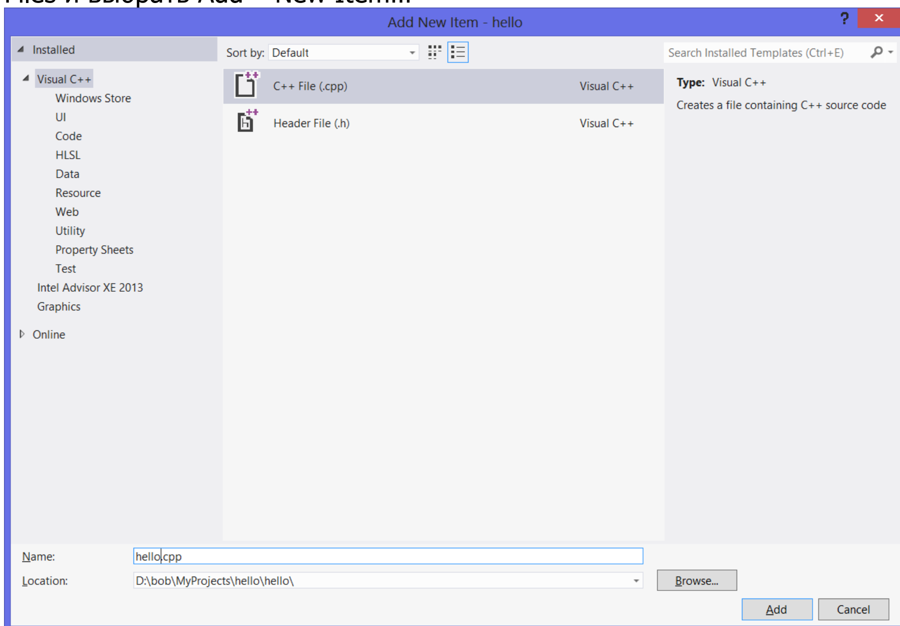



New Project
Recent
Installed
Templates
Visual C++
Windows Store
ATL
CLR
General
MFC
Test
Win32
Windows Phone
Intel(R) Visual Fortran
LightSwitch
Other Languages
Other Project Types
Modeling Projects
Samples
Online
.NET Framework 4.5
Sort by: Default
Search Installed Templates (Ctrl+E)
Win32 Console Application Visual C++
MFC Application Visual C++
Win32 Project Visual C++
Blank App (XAML) Visual C++
Empty Project Visual C++
Grid App (XAML) Visual C++
Split App (XAML) Visual C++
Direct2D App (XAML) Visual C++
Direct3D App Visual C++
DLL (Windows Store apps) Visual C++
Static Library (Windows Store apps) Visual C++
Type: Visual C++
A project for creating a Win32 console application
Name: hello
Location: D:\bob\MyProjects
Solution name: hello
Browse...
Create directory for solution
Add to source control
OK
Cancel


Win32 Application Wizard - hello
Application Settings
Overview
Application Settings
Application type:
Windows application
Console application
DLL
Static library
Add common header files for:
ATL
MFC
Additional options:
Empty project
Export symbols
Precompiled header
Security Development Lifecycle (SDL) checks
< Previous
Next >
Finish
Cancel



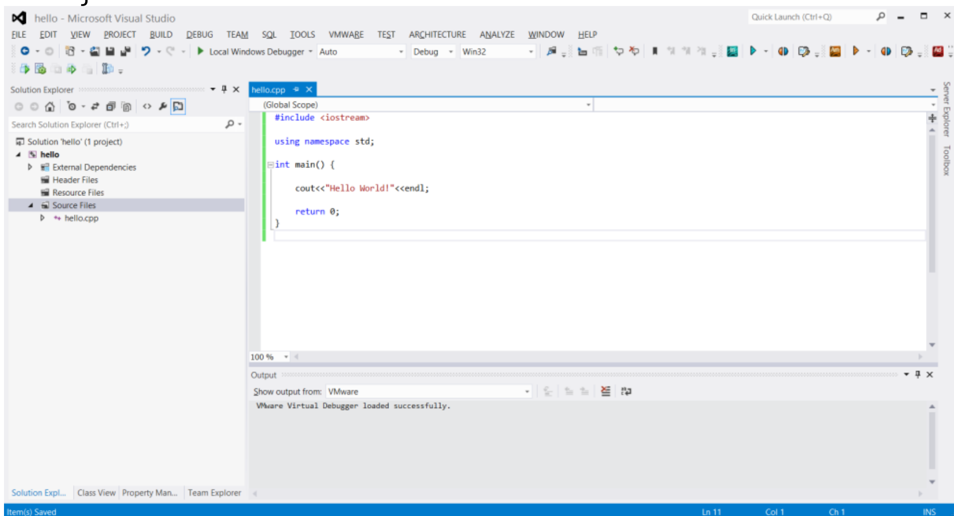

2) В левой части Solution Explorer – hello видны папки, созданные для проекта. Добавить в папку Source Files новый .cpp-файл. Для этого щелкнуть правой кнопкой мыши по папке Source Files и выбрать Add – New Item…




Справа в Templates: выделить C++ File (.cpp), в поле Name: задать имя, например, hello. Нажать на кнопку Add.

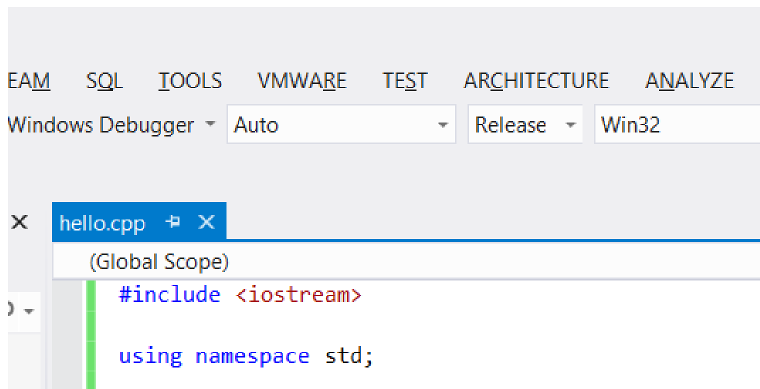
3) В окне редактора набрать приведенный ниже текст (а в дальнейшем изменять его под тексты заданий) и откомпилировать (Build - Compile в меню, или Ctrl+F7).

```
#include <iostream>

using namespace std;

int main() {
	cout<<"Hello World!"<<endl;
	return 0;
	}
```



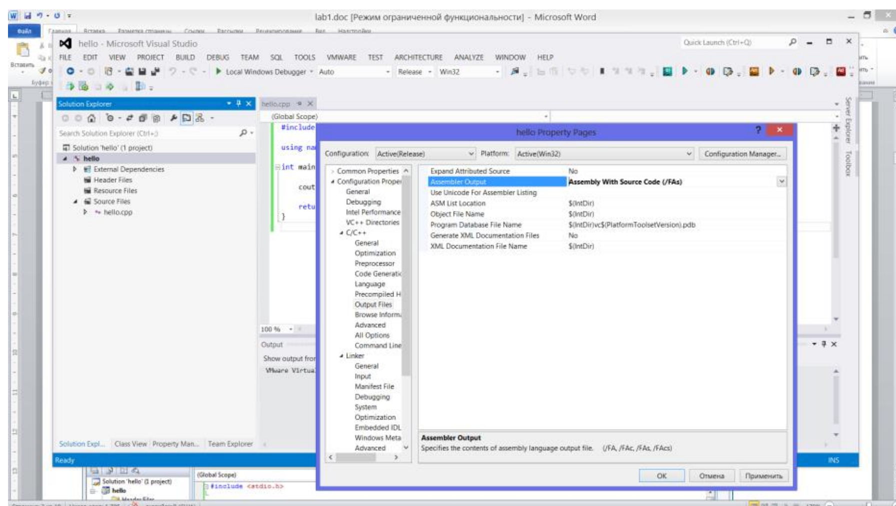

4) Запустить на выполнение (Debug - Start Without Debugging в меню, или Ctrl+F5). Найти в файловой системе полученный выше .exe-файл, скопировать его в другой каталог и запустить в командной строке.

5) Изменить тип проекта на Release,




вновь построить исполняемый модуль (Build – Build Solution в меню, или F7) и сравнить размеры с размерами отладочной версии.

6) Чтобы получить ассемблерный листинг, генерируемый компилятором по C++-коду, нужно в Solution Explorer щелкнуть правой кнопкой мыши по имени проекта hello и на вкладке Configuration Properties – C/C++ – Output Files задать значение Assembler Output – Assembly With Source Code.



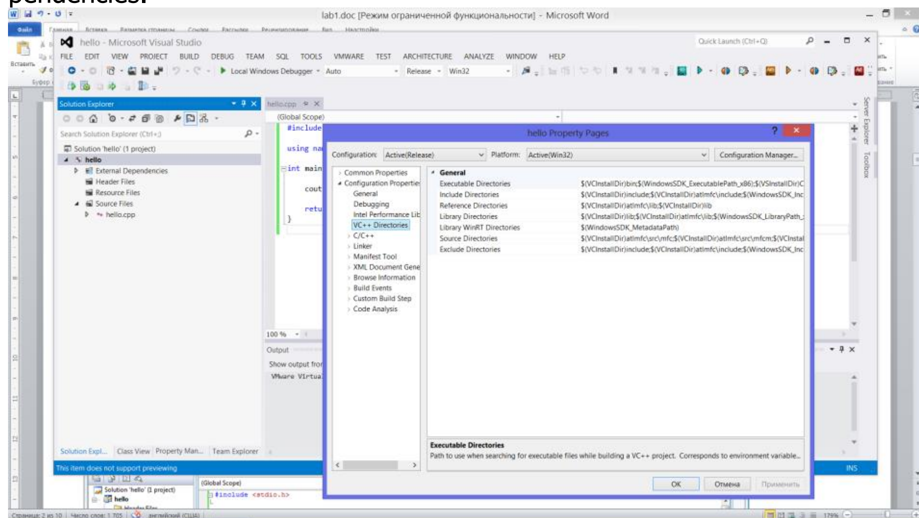

**II. Подключение дополнительных библиотек.**

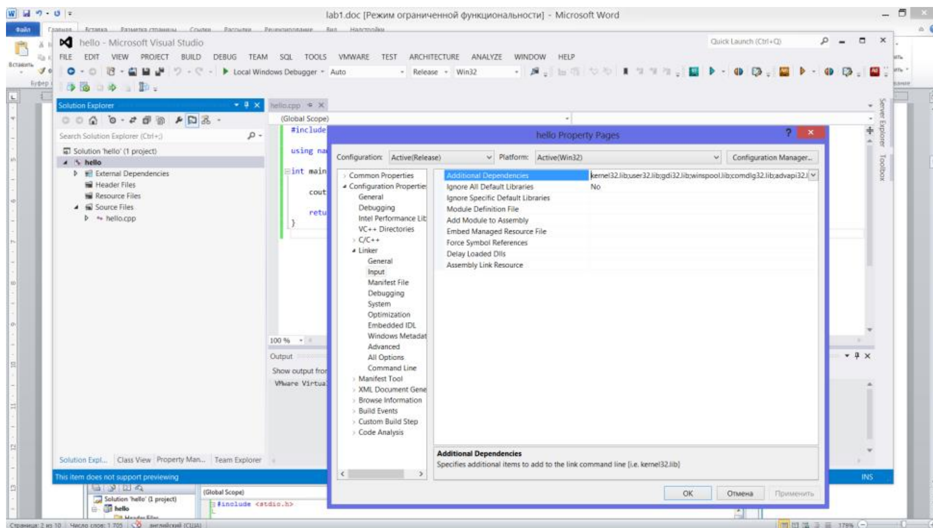
Для использования классов или функций дополнительной (отличной от стандартной библиотеки С++) библиотеки нужно сделать видимыми в программе их описания. Для этого разра-

ботчик библиотеки должен предоставить файлы заголовков. Эти файлы должны быть добавлены в текст программы с помощью директив #include. Можно в самих этих директивах задавать полные пути к заголовочным файлам (не самый удобный вариант), или задавать только имена файлов, но тогда нужно будет указать среде Visual Studio, в каких каталогах их искать. То же самое относится и к самой библиотеке (или нескольким библиотекам). Правда, заголовочные файлы должны быть видны препроцессору, а связывание программы с библиотеками осуществляет компоновщик. В результате нужно сделать следующее:

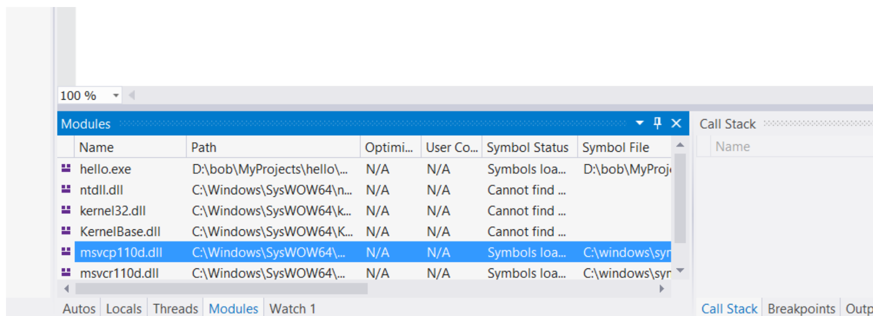
1) В свойствах проекта выбрать вкладку VC++ Directories и добавить пути к папкам с файлами заголовков в категорию Include Directories, а пути к библиотечным файлам в Library Directories.

2) В папках с библиотечными файлами могут располагаться несколько библиотек, поэтому нужно указать конкретные библиотеки, используемые программой в Linker – Input – Additional Dependencies.





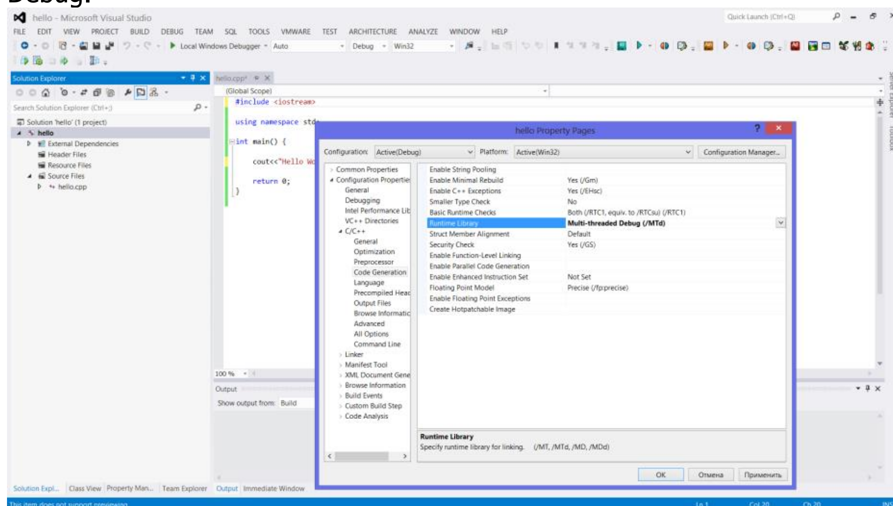

8) По умолчанию при создании программы Visual Studio использует динамическое связывание ее со стандартной бибилиотекой C++. Чтобы увидеть, какие dll-библиотеки использует наша программа нужно выбрать Debug конфигурацию, построить проект и запустить его на отладку – Debug – Start Debuging (F5). Затем открыть окно Debug – Windows – Modules.



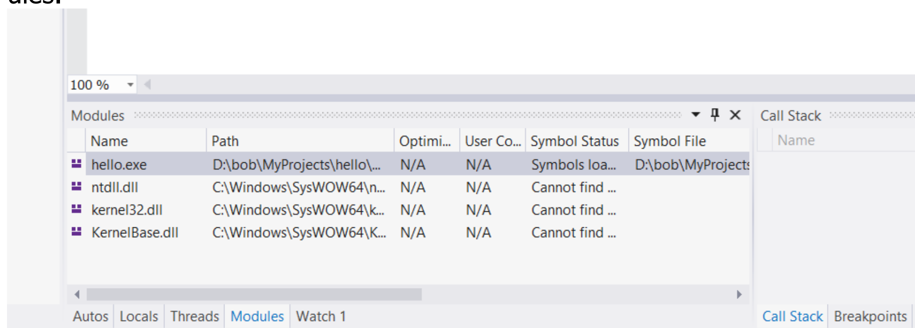

Здесь используются библиотеки msvcp110d.dll – отладочная версия стандартной библиотеки С++ и msvcr110d.dll – библиотеки Си. Если перенести exe-модуль программы на компьютер, на котром не установлены такие бибилотеки, программа не сможет выполняться. Нужно либо переносить их вместе со своей программой, либо отказаться от динамической компоновки в пользу статической. В этом случае коды нужных программе функций стандартной бибилиотеки будут добавлены к exe-модулю про-

граммы и сами библиотеки для запуска программы будут уже не нужны.

Чтобы задать статическую компоновку в свойствах проекта нужно перейти на C++ – Code Generation и в категории Runtime Library вместо Multi-threaded Debug DLL выбрать Multi-threaded Debug.




Постройте проект и запустите его на отладку – Debug – Start Debuging (F5). Затем откройте окно Debug – Windows – Modules.




### III. Стандартный поток вывода.

В C++ стандартный поток вывода представлен глобальной переменной с именем cout. cout – это экземпляр стандартного класса ostream, объявление которого помещается в файле iostream. Глобальная переменная cout также описана в этом файле, поэтому, для использования cout в программе нужно задать директиву препроцессора #in- clude <iostream>. В языке Си

определена операция побитового сдвига влево <<. Например, 2 << 1 – это выражение, в результате вычисления которого двоичное представление числа 2 (10) сдвигается в памяти на один двоичный разряд влево, на освободившееся место заносится ноль, получается двоичное 100, или, в десятичной системе счисления, 4. В С++ операция << переопределена для класса ostream и называется операцией помещения в поток. Переопределение для класса в данном случае означает, что левым (первым) аргументом операции должен быть экземпляр класса ostream (cout для стандартного потока вывода). Каким может быть тип правого (второго) аргумента << определяется при создании класса ostream. Операцию можно переопределять многократно, таким образом, обеспечив возможность задавать в качестве второго аргумента величины разных типов. В классе ostream операция << переопределена для всех стандартных типов С++, целочисленных, вещественных, символов, строк, адресов и для некоторых нестандартных типов, например, в поток можно помещать имена функций (за именем функции скрывается адрес начала расположения кода этой функции в памяти), имеющих один аргумент – поток вывода ostream и возвращающих ссылку на поток ostream. Такие функции называются манипуляторами потока.

Применяя к потоку вывода операцию <<, мы, тем самым, помещаем в поток, заданный первым ее аргументом, значение второго аргумента. Помещение в поток вывода означает по умолчанию вывод значения на экран (в консольное окно). Можно составить выражение, в котором будут заданы несколько операций <<, тогда они выполняются слева – направо, т.е. вначале отображается второй аргумент первой операции <<, затем второй и т.д. Например,

```
cout<<"one="<<1;
```

Выводит на экран вначале строку one=, затем 1 (one=1). При этом целое значение 1 выводится так же, как и в случае использования функции printf("%d",1); стандартной библиотеки Си.

## 1.2. Задание к лабораторной работе

1. Комментарии в С++ как и в Си могут быть многострочными

```
/*
Многострочный комментарий
*/
```

и однострочными // комментарий до конца строки

Закомментируйте строку using namespace std;

Не удаляя комментария, исправьте возникшие ошибки.

2. Используйте стандартный вывод, чтобы установить, какие арифметические операции для целых чисел можно заменить операциями сдвига влево (<<) и вправо (>>).

### 1.3. Контрольные вопросы

1. Как создать консольный C++-проект в MS Visual Studio?
2. В чем разница между Debug и Release версиями проекта? Как создать Release версию проекта?
3. Как создать исполняемый модуль для проекта? Как запустить его на выполнение?
4. Как получить ассемблерный листинг программы?
5. В разница между статической и динамической компоновкой?
6. Как подключить к проекту дополнительную динамическую библиотеку?
7. Как реализован стандартный поток вывода в C++?

### 1.4. Литература

1. Brian W. Kernighan, Dennis M. Ritchie. "The C programming language". Second edition – Prentice Hall. – 1988, 272 p.

2. Стэнли Б. Липпман. "Язык программирования C++. Вводный курс". – М.: Вильямс. – 2007, 896 с.

3. Г. Шилдт. "Самоучитель C++". – СПб.: БХВ-Петербург. – 2001, 688 с.

4. К.Арнольд, Дж.Гослинг, Д.Холмс. "Язык программирования Java". 3-е изд. – М.: Вильямс. – 2001, 624 с.

# 2. ЛАБОРАТОРНАЯ РАБОТА №2: ОПЕРАЦИИ И ВЫРАЖЕНИЯ В C/C++

### 2.1. Теория

**Выражения.** Выражение - это конструкция языка, предназначенная для получения некоторого значения. Тип вычисленного значения определяет тип выражения. Выражение может содержать константы, переменные, знаки операций, вызовы функций, возвращающих значения и круглые скобки. Выражение не обязательно должно содержать все выше перечисленные конструкции,

например, одна константа или одна переменная являются частными случаями выражений.

**Операции.** Операции являются элементами выражений и также возвращают значение. Помимо собственно действия выполняемого операцией по вычислению значения, с каждой операцией связаны следующие атрибуты: арность (количество аргументов), знак операции, ее приоритет и ассоциативность. Знак операции не обязательно уникален, т.е. две операции могут иметь одно и то же обозначение. Уникальным должно быть сочетание знака, арности, порядка следования значка операции и аргумента тогда компилятор сможет однозначно распознать операцию. Если выражение содержит одну операцию, например, x = 7, то ни приоритет, ни ассоциативность неважны. Если же в выражении несколько операций, компилятору необходимо знать, в каком порядке их выполнять. Например, в выражении x + 2 * y прежде должна выполняться операция умножения и, затем, операция сложения. Чтобы это было так, операция умножения должна иметь более высокий приоритет, нежели операция сложения. Для изменения порядка выполнения операций используются круглые скобки. Ассоциативность важна, если несколько операций в выражении имеют одинаковый приоритет, например, x = y = x = 0. В этом примере операции должны выполняться справа налево (правая ассоциативность). Унарные операции, условная операция и операции присваивания выполняются справа налево, остальные — слева направо. Оператор вида ?= существует для любой арифметической, логической и побитовой операции ?, допустимой в Си. Например, для арифметических операций +, -, *, /, % можно использовать операции += увеличить на -= уменьшить на *= домножить на /= поделить на %= поделить с остатком на к примеру, строка x *= 2.0; удваивает значение вещественной переменной x. Операторы вида ?= можно использовать даже для операций ?, которые записываются двумя символами. Например, операции логического умножения и сложения записываются в Си как && (двойной амперсенд) и || (двойная вертикальная черта). Соответственно, логические операторы AND и OR записываются в виде &&= и ||=, например,

**Пример** bool x, y; x &&= y; // эквивалентно x = x && y; x ||= y; // эквивалентно x = x || y;

**Пример** // не правильное выражение int value = 5; cout << value += 3; //(ОШИБКА) передаем в поток вывода значение пере-

менной value + 3 Компилятор не поймет выражения в строке 2, так как операция сдвига влево имеет больший приоритет нежели операция суммирования с присваиванием. В итоге программу с таким выражением даже нельзя будет запустить. Суть в том, что компилятор понимает это выражение не так как мы, а совсем по-другому. Как будет рассуждать компилятор: «В выражении две операции, причѐм первая операция << имеет больший приоритет, чем вторая +=, значит передам сначала в поток вывода значение переменной value, а потом прибавлю 3 к ?????? А не к чему прибавить 3, так как переменная value передаѐтся в поток вывода.» Вот в этом и заключается ошибка, а чтобы еѐ не было необходимо просто поставить круглые скобки. 1 // правильное выражение 2 int value = 5; 3 cout << (value += 3); // передаем в поток вывода значение переменной value + 3 В таком случае, сначала выполнится выражение в круглых скобках, а потом значение переменной value передастся в поток вывода.

**Пример**

```
int x = 5, y = 5, a, b;
a = (++x) + 2; // переменной a присваивается значение 8
b = (y++) + 2; // переменной b присваивается значение 7
```

**Пример**

```
int x, y;
x = 5;
y = x++;
cout<<"x="<<x<<"\ty="<<y<<endl;
x = 5;
y = ++x;
cout<<"x="<<x<<"\ty="<<y<<endl;
```

*Замечание:* У всякой операции есть прямой эффект - возвращаемое ею значение. У некоторых операций есть еще и побочный эффект, например, у инкремента/декремента прямой эффект разный в зависимости от вида операции(префиксная или постфиксная), а побочный эффект (изменение значения аргумента операции) один и тот же для обоих видов.

**Пример**

```
int value = 2011;
cout << "value = " << value << endl; // начальное значение
cout << "++value = " << ++value << endl; // операция преинкремента
```

cout << "value++ = " << value++ << endl; // операция постинкремента

cout << "value = " << value << endl; /* конечное значение в переменной value после выполнения операции постинкремента */

cout << "--value = " << --value << endl; // операция предекремента

cout << "value-- = " << value-- << endl; // операция постдекремента

cout << "value = " << value << endl; /* конечное значение в переменной value после выполнения операции постдекремента */

**Логические операции**

Существует три логические операции:

Логическая операция И &&, нам уже известная;

Логическая операция ИЛИ ||;

Логическая операция НЕ ! или логическое отрицание.

Для Си/C++ любое ненулевое значение может рассматриваться как истина, а нулевое как ложь. Например, cout<<(2<1)<<endl; // выводит 0 Логические операции образуют сложное (составное) условие из нескольких простых (два или более) условий. **И &&** a == 3 && b > 4 Составное условие истинно, если истинны оба простых условия **ИЛИ ||** a == 3 || b > 4 Составное условие истинно, если истинно, хотя бы одно из простых условий **НЕ !** !( a == 3) Условие истинно, если a не равно 3

**Пример**

bool a1 = true, a2 = false; // объявление логических переменных

bool a3 = true, a4 = false;

cout << "Tablica istinnosti log operacii &&" << endl;

cout << "true && false: " << ( a1 && a2 ) << endl // логическое И

<< "false && true: " << ( a2 && a1 ) << endl

<< "true && true: " << ( a1 && a3 ) << endl

<< "false && false: " << ( a2 && a4 ) << endl;

cout << "Tablica istinnosti log operacii ||" << endl;

cout << "true || false: " << ( a1 || a2 ) << endl // логическое ИЛИ

<< "false || true: " << ( a2 || a1 ) << endl

```
<< "true || true: " << ( a1 || a3 ) << endl
<< "false || false: " << ( a2 || a4 ) << endl;
cout << "Tablica istinnosti log operacii !" << endl;
cout << "!true: " << ( !a1 ) << endl // логическое НЕ
<< "!false: "<< ( !a2 ) << endl;
```

**Операции сравнения**

Операция сравнения сравнивает два выражения. В результате вырабатывается логическое значение - true или false (истина или ложь) в зависимости от значений выражений. Примеры: bool res; int x, y; res = (x == y); // true, если x равно y, иначе false res = (x == x); // всегда true res = (2 < 1); // всегда false Операции сравнения в Си обозначаются следующим образом: == равно, != не равно, > больше, >= больше или равно, < меньше, <= меньше или равно.

**Пример**

```
bool res;
int x, y;
res = (x == y); // true, если x равно y, иначе false
res = (x == x); // всегда true
res = (2 < 1); // всегда false
```

**Побитовые операции**

Существует три побитовые операции:
побитовое И, обозначение: &
побитовое ИЛИ, обозначение: |
побитовое исключающее ИЛИ, обозначение: ^

Данные операции работают с битами ячеек памяти, причѐм операнды и результат могут быть заданы в другой форме, например, в десятичной.

В основном побитовые операции применяются для манипуляций с битовыми масками. Например, пусть целое число x описывает набор признаков некоторого объекта, состоящий из четырех признаков. Назовем их условно A, B, C, D. Пусть за признак A отвечает нулевой бит слова x (биты в двоичном представлении числа нумеруются справа налево, начиная с нуля). Если бит равен единице (говорят бит установлен), то считается, что объект обладает признаком A. За признаки B, C, D отвечают биты с номерами 1, 2, 3. Общепринятая практика состоит в том, чтобы определить константы, отвечающие за соответствующие признаки (их обычно называют масками ):

```
const int MASK_A = 1;
const int MASK_B = 2;
const int MASK_C = 4;
const int MASK_D = 8;
```

Эти константы содержат единицу в соответствующем бите и нули в остальных битах. Для того чтобы проверить, установлен ли в слове x бит, соответствующий, к примеру, признаку D, используется операция побитового логического умножения. Число x умножается на константу MASK_D ; если результат отличен от нуля, то бит установлен, т.е. объект обладает признаком D, если нет, то не обладает.

Такая проверка реализуется следующим фрагментом:

```
if ((x & MASK_D) != 0) {
// Бит D установлен в слове x, т.е.
// объект обладает признаком D
. . .
} else {
// Объект не обладает признаком D
. . .
}
```

При побитовом логическом умножении константа MASK_D обнуляет все биты слова x, кроме бита D, т.е. как бы вырезает бит D из x. В двоичном представлении это выглядит примерно так:

```
x: 0101110110...10*101
MASK_D: 0000000000...001000
x & MASK_D: 0000000000...00*000
```

Звездочкой здесь обозначено произвольное значение бита D слова x.

Для установки бита D в слове x используется операция побитового логического сложения: x = (x | MASK_D); // Установить бит D в слове x

Чаще это записывается с использованием операции |= типа "увеличить на":

```
x |= MASK_D; // Установить бит D в слове x
```

В двоичном виде это выглядит так:

```
x: 0101110110...10*101
MASK_D: 0000000000...001000
x | MASK_D: 0101110110...101101
```

Операция побитового отрицания " ~ " инвертирует биты слова:

```
x: 0101110110...101101
```

~x: 1010001001...010010

Для очистки (т.е. установки в ноль) бита D используется комбинация операций побитового отрицания и побитового логического умножения:

x = (x & ~MASK_D); // Очистить бит D в слове x

или, применяя операцию " &= " типа "домножить на":

x &= ~MASK_D; // Очистить бит D в слове x

Здесь сначала инвертируется маска, соответствующая биту D,

MASK_D: 0000000000...001000

~MASK_D: 1111111111...110111

в результате получаются единицы во всех битах, кроме бита D. Затем слово x побитно домножается на инвертированную маску:

x: 0101110110...10*101

~MASK_D: 1111111111...110111

x & ~MASK_D: 0101110110...100101

В результате в слове x бит D обнуляется, а остальные биты остаются неизменными.

Побитовую операцию ^ называют сложением по модулю 2, а также "исключающим или". Часто для нее используется аббревиатура XOR, от eXclusive OR. "Таблица сложения" для этой операции выглядит следующим образом:

0 ^ 0 = 0, 0 ^ 1 = 1,

1 ^ 0 = 1, 1 ^ 1 = 0.

Пусть x - произвольное целое число, m - маска, т.е. число, в котором интересующие программиста биты установлены в единицу, остальные в ноль. В результате выполнения операции XOR

x = (x ^ m); или, в более удобной записи, x ^= m;

биты в слове x, соответствующие установленным в единицу битам маски m, изменяются на противоположные (инвертируются). Биты слова x, соответствующие нулевым битам маски, не меняют своих значений.

**Пример:**

x: 101101...1001011110

m: 000000...0011111100

x ^ m: 101101...1010100010

Операция XOR обладает замечательным свойством: если дважды прибавить к слову x произвольную маску m, то в результате получается исходное значение x:

((x ^ m) ^ m) == x

Прибавление к слову x маски m можно трактовать как шифрование x, ведь в результате биты x, соответствующие единичным битам маски m, инвертируются. Если маска достаточно случайная, то в результате x тоже принимает случайное значение. Процедура расшифровки в данном случае совпадает с процедурой шифрования и состоит в повторном прибавлении маски m.

**Операции сдвига**

Операции сдвига применяются к целочисленным переменным: двоичный код числа сдвигается вправо или влево на указанное количество позиций. Сдвиг вправо обозначается двумя символами "больше" >>, сдвиг влево - двумя символами "меньше" <<. Примеры:

```
int x, y;
. . .
x = (y >> 3); // Сдвиг на 3 позиции вправо
y = (y << 2); // Сдвиг на 2 позиции влево
```

При сдвиге влево на k позиций младшие k разрядов результата устанавливаются в ноль. Сдвиг влево на k позиций эквивалентен умножению на число 2k.

Сдвиг вправо более сложен, он по-разному определяется для беззнаковых и знаковых чисел. При сдвиге вправо беззнакового числа на k позиций освободившиеся k старших разрядов устанавливаются в ноль. Например, в двоичной записи имеем:

```
unsigned x;
x = 110111000...10110011
x >> 3 = 000110111000...10110
```

Сдвиг вправо на k позиций соответствует целочисленному делению на число 2k.

При сдвиге вправо чисел со знаком происходит так называемое "расширение знакового разряда". Именно, если число неотрицательно, т.е. старший, или знаковый, разряд числа равен нулю, то происходит обычный сдвиг, как и в случае беззнаковых чисел. Если же число отрицательное, т.е. его старший разряд равен единице, то освободившиеся в результате сдвига k старших разрядов устанавливаются в единицу. Число, таким образом, остается отрицательным. При k = 1 это соответствует делению на 2 только для отрицательных чисел, не равных -1. Для числа -1, все биты двоичного кода которого равны единице, сдвиг вправо не приводит к его изменению. Пример (используется двоичная запись):

```
int x;
```

x = 110111000...10110011
x >> 3 = 111110111000...10110

В программах лучше не полагаться на эту особенность сдвига вправо для знаковых чисел и использовать конструкции, которые заведомо одинаково работают для знаковых и беззнаковых чисел. Например, следующий фрагмент кода выделяет из целого числа составляющие его байты и записывает их в целочисленные переменные x0, x1, x2, x3, младший байт в x0, старший в x3. При этом байты трактуются как неотрицательные числа. Фрагмент выполняется одинаково для знаковых и беззнаковых чисел:

```
int x;
int x0, x1, x2, x3;
. . .
x0 = (x & 255);
x1 = ((x >> 8) & 255);
x2 = ((x >> 16) & 255);
x3 = ((x >> 24) & 255);
```

Здесь число 255 играет роль маски. При побитовом умножении на эту маску из целого числа вырезается его младший байт, поскольку маска 255 содержит единицы в младших восьми разрядах. Чтобы получить байт числа x с номером n, n = 0,1,2,3, сначала сдвигаем двоичный код x вправо на 8n разрядов, таким образом, байт с номером n становится младшим. Затем с помощью побитового умножения вырезается младший байт.

### Операция sizeof

Переменная одного и того же типа на разных платформах может занимать различное число байтов памяти. Язык Си предоставляет программисту возможность получить размер элемента данного типа или размер переменной в байтах, для этого служит оператор sizeof. Аргумент sizeof указывается в круглых скобках, он может быть типом или переменной. Рассмотрим несколько примеров. Пусть определены следующие переменные: int i; char c; short s; long l; double d; float f; bool b; Тогда приведенные ниже выражения в 32-разрядной архитектуре имеют следующие значения:

<table>
<tr><th>Приоритеты и ассоциативность операций. P - уровень приоритета, Ac - ассоциативность, Ap - арность.<br>____________<br>____________<br>__ _____ P</th><th>Знак операции</th><th>Ac</th><th>Ap</th><th>Примечание</th></tr>
<tr><td>15</td><td>++ -- [] () . -></td><td>·</td><td>1/2</td><td>постинкремент, постдекремент, индексирование, вызов функции, доступ к полю структуры (арность 2), доступ к полю структуры через указатель (арность 2)</td></tr>
<tr><td>14</td><td>sizeof ++ -- & * + - ~ ! (тип)</td><td>·</td><td>1</td><td>размер в байтах, преинкремент, предекремент, получение адреса, разадресация, смена знака, смена знака, побитовое отрицание, логическое отрицание, приведение типа</td></tr>
<tr><td>13</td><td>* / %</td><td>·</td><td>2</td><td>умножение, деление, остаток от целочисленного деления</td></tr>
<tr><td>12</td><td>+ -</td><td>·</td><td>2</td><td>сложение, вычитание</td></tr>
<tr><td>11</td><td>>> <<</td><td>·</td><td>2</td><td>побитовый сдвиг вправо, влево</td></tr>
<tr><td>10</td><td><<=>>=</td><td>·</td><td>2</td><td>проверка на меньше, меньше или равно, больше, больше или равно</td></tr>
<tr><td>9</td><td>== !=</td><td>·</td><td>2</td><td>проверка на равенство, не равенство</td></tr>
<tr><td>8</td><td>&</td><td>·</td><td>2</td><td>побитовая конъюнкция (И)</td></tr>
<tr><td>7</td><td>^</td><td>·</td><td>2</td><td>побитовое исключающее или</td></tr>
<tr><td>6</td><td>|</td><td>·</td><td>2</td><td>побитовая дизъюнкция (ИЛИ)</td></tr>
<tr><td>5</td><td>&&</td><td>·</td><td>2</td><td>конъюнкция (логическое И)</td></tr>
<tr><td>4</td><td>||</td><td>·</td><td>2</td><td>дизъюнкция (логическое ИЛИ)</td></tr>
</table>

| 3 | ?: | · | 3 | операция условия |
|---|---|---|---|---|
| 2 | = += -= *= /= %= &= \|= ^= <<= >>= | · | 2 | операции присваивания |
| 1 | , | · | 2 | операция запятая |

## 2.2. Задание к лабораторной работе

**ЗАДАНИЕ 1.** Определите, какие из приведенных ниже конструкций являются выражениями.

1) x + 1.5
2) sin( x )
3) y = x +1;
4) 127
5) x = y = z = 0
6) { z = 2 * cos( x ) - y; }
7) int x = 5;

**ЗАДАНИЕ 2.** Определите, какими будут значения переменных i, j, k после вычисления значения следующего выражения:

k = ( --i + 2 * j - k++ , j-- + i - k )

если изначально эти значения были следующими:

i = 5 j = 12 k = 7 Объясните результат.

**ЗАДАНИЕ 3.** Напишите программу, вычисляющую значение следующего выражения:

*m* = i /16 + j *128-17 + k *2- l /32 , если i > j + 2 * k и *m* = i /16 + j *128-17 + k *2 + l /32 в противном случае, для произвольных значений переменных i, j, k и l, не используя при этом операции умножения и деления.

**ЗАДАНИЕ 4.** Организовать ввод двух целочисленных чисел. Вывести их в шестнадцатеричном виде. Выполнить побитовое сложение и умножение данных чисел, побитовые сдвиги влево и вправо. Вывести результаты в шестнадцатеричном виде и объяснить их.

**ЗАДАНИЕ 5.** Реализовать алгоритм XOR-шифрования одного символа. Рассмотреть беззнаковое целое как четыре упакованных символа (байта). Задать можно как шестнадцатеричное значение, например, 0x41424344 как сжатое ABCD. Тогда можно применить XOR (с 32-битовой маской) к целому значению (блоку

из четырех символов). С помощью побитовых операций вывести отдельные байты (printf, спецификация %c - printf("%c%c%c%c\n",u>>24,(u & 0xffffff)>>16,(u & 0xffff)>>8,(u & 0xff)); ) до шифрования и после (или после шифрования и после расшифрования).

**ЗАДАНИЕ 6.** Напишите программу, которая меняет местами последний (младший) и предпоследний байты переменной i типа int.

**ЗАДАНИЕ 7.** Напишите программу, которая определяет, сколько единиц содержится в двоичном представлении переменной типа char.

**ЗАДАНИЕ 8.** Реализовать битовую маску: представить, что память состоит из 8 блоков. Битовая карта описывает занят (1) или свободен (0) блок, т.е. для описания 8 блоков достаточно одного байта. Спросить у пользователя, какие блоки занять, вывести битовую карту, потом какие освободить и также вывести битовую карту.

### 2.3. Контрольные вопросы

1. Что такое выражение?
2. Что такое ассоциативность?
3. В чем различие префиксной и постфиксной операций инкремента и декремента?
4. Что является результатом логической операции? Какие существуют логические операции?
5. Какие существуют побитовые операции?
6. Для чего используются битовые маски?
7. Что получается в результате применения операции битового сдвига?
8. Для чего используется оператор sizeof?
9. Напишите таблицу умножения для операции OR (AND, XOR).
10. Как сбросить 3 бит некоторого целого значения?
11. Как установить второй и третий биты целого значения?
12. Некоторое целое значение зашифровано с помощью операции XOR и маски
13. 0x12345678. Как расшифровать это значение?
14. Какая запись правильная

cout<<(x*=5); или cout<<x*=5; ?

15. Что будет выведено на экран в результате выполнения следующих действий

cout<<(sizeof(int)>sizeof(long)); ?

16. Дана целая беззнаковая переменная u. Значение какого байта будет выведено на экран cout<<((u & 0xffff)>>8); ?

### 2.4. Литература

1. Brian W. Kernighan, Dennis M. Ritchie. "The C programming language". Second edition – Prentice Hall. – 1988, 272 p.

2. Стэнли Б. Липпман. "Язык программирования C++. Вводный курс". – М.: Вильямс. – 2007, 896 с.

3. Г. Шилдт. "Самоучитель C++". – СПб.: БХВ-Петербург. – 2001, 688 с.

4. К.Арнольд, Дж.Гослинг, Д.Холмс. "Язык программирования Java". 3-е изд. – М.: Вильямс. – 2001, 624 с.

# 3. ЛАБОРАТОРНАЯ РАБОТА №3: ОПЕРАТОРЫ C/C++

### 3.1. Теория

Выделяются следующие операторы:

1) ;
2) {}
3) *выражение;*
4) if
5) for
6) while
7) do while
8) break;
9) continue;
10) switch*;*
11) return;
12) goto.

**Условные операторы**

Условные операторы позволяют выбрать один из вариантов выполнения действий в зависимости от каких-либо условий.

Условие – это выражение, которое может иметь логический тип (true (истина) или false (ложь)) или целочисленный (веще-

ственный) или любой тип, который можно привести к числу.

**Оператор if** выбирает один из двух вариантов последовательности вычислений.

```
if (выражение)
оператор1
else
оператор2
```

Если выражение истинно, выполняется оператор1, если ложно, то выполняется оператор2.

```
if (x > y)
a = x;
else
a = y;
```

В данном примере переменной a присваивается значение максимума из двух величин x и y.

Конструкция else необязательна. Можно записать:

```
if (x < 0)
x = -x;
abs = x;
```

В данном примере оператор x = -x; выполняется только в том случае, если значение переменной x было отрицательным. Присваивание переменной abs выполняется в любом случае. Таким образом, приведенный фрагмент программы изменит значение переменной x на его абсолютное значение и присвоит переменной abs новое значение x.

Если в случае истинности выражения необходимо выполнить несколько операторов, их можно заключить в фигурные скобки:

```
if (x < 0) {
x = -x;
cout << "Изменить значение x на противоположное по знаку";
}
abs = x;
```

Теперь если x отрицательно, то не только его значение изменится на противоположное, но и будет выведено соответствующее сообщение. Фактически, заключая несколько операторов в фигурные скобки, мы сделали из них один сложный оператор или блок. Прием заключения нескольких операторов в блок работает везде, где нужно поместить несколько операторов вместо одного.

**Пример условия в виде целочисленного и вещественного выражения**

```
int main(int argc, char *argv[])
{
int x=2;
if(-100) cout<<-100<<endl;
if (-20+10) cout<<-20+10<<endl;
if ('a') cout<<'a'<<endl;
if (x/3) cout<<x<<endl;
if (2.3) cout<<2.3<<endl;
system("PAUSE");
return 0;
}
```

Условный оператор можно расширить для проверки нескольких условий:

```
if (x < 0)
cout << "Отрицательная величина";
else if (x > 0)
cout << "Положительная величина";
else
cout << "Ноль";
```

Конструкций else if может быть несколько.

Хотя любые комбинации условий можно выразить с помощью оператора if, довольно часто запись становится неудобной и запутанной.

Оператор выбора **switch** используется, когда для каждого из нескольких возможных значений выражения нужно выполнить определенные действия. Например, предположим, что в переменной code хранится целое число от 0 до 2, и нам нужно выполнить различные действия в зависимости от ее значения:

```
switch (code) {
case 0:
cout << "код ноль";
x = x + 1;
break;
case 1 :
cout << "код один";
y = y + 1;
break;
case 2:
cout << "код два";
z = z + 1;
break;
```

```
default:
cout << "Необрабатываемое значение";
}
```

В зависимости от значения code управление передается на одну из меток case. Выполнение оператора заканчивается по достижении либо оператора break, либо конца оператора switch. Таким образом, если code равно 1, выводится " код один ", а затем переменная y увеличивается на единицу. Если бы после этого не стоял оператор break, то управление "провалилось" бы дальше, была бы выведена фраза " код два ", и переменная z тоже увеличилась бы на единицу.

Если значение переключателя не совпадает ни с одним из значений меток case, то выполняются операторы, записанные после метки default. Метка default может быть опущена, что эквивалентно записи:

```
default:
 ; // пустой оператор, не выполняющий никаких действий
```

Очевидно, что приведенный пример можно переписать с помощью оператора if:

```
if (code == 0) {
cout << "код ноль";
x = x + 1;
} else if (code == 1) {
cout << "код один";
y = y + 1;
} else if (code == 2) {
cout << "код два";
z = z + 1;
} else {
cout << "Необрабатываемое значение";
}
```

Пожалуй, запись с помощью оператора переключения switch более наглядна. Особенно часто переключатель используется, когда значение выражения имеет тип набора.

В C++ есть условная операция, очень похожая на условный оператор выбора if else. Условная операция «? :» называется тернарной операцией (то есть трѐхместная (имеет три операнда), единственная в C++).

**Операторы цикла**

Предположим, нам нужно вычислить сумму всех целых чисел от 0 до 100. Для этого воспользуемся оператором цикла for:

```
int sum = 0;
int i;
for (i = 1; i <= 100; i = i + 1) // заголовок цикла
sum = sum + i; // тело цикла
```

Оператор цикла состоит из заголовка цикла и тела цикла. Тело цикла – это оператор, который будет повторно выполняться (в данном случае – увеличение значения переменной sum на величину переменной i ). Заголовок – это ключевое слово for, после которого в круглых скобках записаны три выражения, разделенные точкой с запятой. Первое выражение вычисляется один раз до начала выполнения цикла. Второе – это условие цикла. Тело цикла будет повторяться до тех пор, пока условие цикла истинно. Третье выражение вычисляется после каждого повторения тела цикла.

Оператор for реализует фундаментальный принцип вычислений в программировании – итерацию. Тело цикла повторяется для разных, в данном случае последовательных, значений переменной i. Повторение иногда называется итерацией. Мы как бы проходим по последовательности значений переменной i, выполняя с текущим значением одно и то же действие, тем самым постепенно вычисляя нужное значение. С каждой итерацией мы подходим к нему все ближе и ближе. С другим принципом вычислений в программировании – рекурсией – мы познакомимся в разделе, описывающем функции.

Пример

```
#include <stdio.h>
int main(void) {
int a = 5, b = 10, c = 25, d = 100, result;
for(int i = 0; i <= d; i += 10) {
printf("%d ", i);
}
printf("\n");
return 0;
}
```

Любое из трех выражений в заголовке цикла может быть опущено (в том числе и все три). То же самое можно записать следующим образом:

```
int sum = 0;
int i = 1;
for (; i <= 100; ) {
sum = sum + i;
```

```
i = i + 1;
}
```

Заметим, что вместо одного оператора цикла мы записали несколько операторов, заключенных в фигурные скобки – блок.

Другой вариант:

```
int sum = 0;
int i = 1;
for (; ;) {
if (i > 100)
break;
sum = sum + i;
i = i + 1;
}
```

В последнем примере мы опять встречаем оператор break. **Оператор break** завершает выполнение цикла. Еще одним вспомогательным оператором при выполнении циклов служит **оператор продолжения continue**. Оператор continue заставляет пропустить остаток тела цикла и перейти к следующей итерации (повторению).

Например, если мы хотим найти сумму всех целых чисел от 0 до 100, которые не делятся на 7, можно записать это так:

```
int sum = 0;
for (int i = 1; i <= 100; i = i+1) {
if ( i % 7 == 0)
continue;
sum = sum + i; }
```

Еще одно полезное свойство цикла for: в первом выражении заголовка цикла можно объявить переменную. Эта переменная будет действительна только в пределах цикла.

Другой формой оператора цикла является **оператор while**. Его форма следующая:

```
while (условие)
оператор
```

Условие – как и в условном операторе if – это выражение, которое принимает логическое значение "истина" или "ложь". Выполнение оператора повторяется до тех пор, пока значением условия является true (истина). Условие вычисляется заново перед каждой итерацией. Подсчитать, сколько десятичных цифр нужно для записи целого положительного числа N, можно с помощью следующего фрагмента:

```
int digits =0;
while (N >= 1) {
```

```
digits = digits + 1;
N = N / 10;
}
```

**Пример**

```
#include <stdio.h>
int main(void) {
int a = 5, b = 10, c = 25, d = 100, result;
while (a <= d) {
if (a % 10 == 0)
printf("%d ", a);
a += 5;
}
printf("\n");
return 0;
}
```

Третьей формой оператора цикла является **цикл do while**. Он имеет форму:

```
do { операторы } while ( условие);
```

Отличие от предыдущей формы цикла while заключается в том, что условие проверяется после выполнения тела цикла. Предположим, требуется прочитать символы с терминала до тех пор, пока не будет введен символ "звездочка".

```
char ch;
do {
ch = getch(); // функция getch возвращает
// символ, введѐный с
// клавиатуры
} while (ch != '*');
```

Пример чтения символов с терминала можно переписать в виде:

```
char ch;
ch = getch();
while (ch != '*') {
ch = getch();
}
```

Для do while разрешена форма без фигурных скобок, если тело цикла состоит из одного оператора, т.е., например,

```
int i = 0;
do ++i;
while(i<10);
```

Пример

```
#include <stdio.h>
int main(void) {
int a = 5, b = 10, c = 25, d = 100, result;
do {
printf("%d ", a);
} while (a >= d);
printf("\n");
return 0;
}
```

**Одномерные массивы в С++**

Одномерный массив — массив, с одним параметром, характеризующим количество элементов одномерного массива. Фактически одномерный массив — это массив, у которого может быть только одна строка, и n-е количество столбцов. Столбцы в одномерном массиве — это элементы массива. На рисунке 1 показана структура целочисленного одномерного массива a. Размер этого массива — 16 ячеек.

Рисунок 1 — Массивы в С++

Заметьте, что максимальный индекс одномерного массива a равен 15, но размер массива 16 ячеек, потому что нумерация ячеек массива всегда начинается с 0.

Индекс ячейки – это целое неотрицательное число, по которому можно обращаться к каждой ячейке массива и выполнять какие-либо действия над ней (ячейкой).

//пример объявления одномерного массива, изображенного на рисунке 1:

int a[16];

где, int - целочисленный тип данных; a - имя одномерного массива; 16 — размер одномерного массива, 16 ячеек.

Всегда сразу после имени массива идут квадратные скобочки, в которых задаѐтся размер одномерного массива, этим массив и отличается от всех остальных переменных.

//ещѐ один способ объявления одномерных массивов

int mas[10], a[16];

Объявлены два одномерных массива mas и a размерами 10 и 16 соответственно. Причѐм в таком способе объявления все массивы будут иметь одинаковый тип данных, в нашем случае - int.

// массивы могут быть инициализированы при объявлении:

int a[16] = { 5, -12, -12, 9, 10, 0, -9, -12, -1, 23, 65, 64, 11, 43, 39, -15 }; // инициализация одномерного массива

Инициализация одномерного массива выполняется в фигурных скобках после знака равно, каждый элемент массива отделяется от предыдущего запятой.

int a[]={5,-12,-12,9,10,0,-9,-12,-1,23,65,64,11,43,39,-15}; // инициализации массива без определения его размера.

В данном случае компилятор сам определит размер одномерного массива. Размер массива можно не указывать только при его инициализации, при обычном объявлении массива обязательно нужно указывать размер массива. Разработаем простую программу на обработку одномерного массива.

**Пример**

```
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
cout << "obrabotka massiva" << endl;
int array1[16] = { 5, -12, -12, 9, 10, 0, -9,
-12, -1, 23, 65, 64, 11, 43, 39, -15 }; // объявление и инициализация одномерного массива
cout << "indeks" << "\t\t" << "element massiva" << endl; // печать заголовков
for (int counter = 0; counter < 16; counter++) //начало цикла
{
//вывод на экран индекса ячейки массива, а затем содержимого этой ячейки
cout << "array1[" << counter << "]" << "\t\t" << array1[counter] << endl;
}
system("pause");
return 0;
}
```

**Пример**

Программа должна последовательно считывать десять введѐнных чисел с клавиатуры. Все введѐнные числа просуммировать, результат вывести на экран.

```
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
int array1[10]; // объявляем целочисленный массив
```

```
cout << "Enter elementi massiva: " << endl;
int sum = 0;
for ( int counter = 0; counter < 10; counter++ ) // цикл для считывания чисел
cin >> array1[counter]; // считываем вводимые с клавиатуры числа
cout << "array1 = {";
for ( int counter = 0; counter < 10; counter++ ) // цикл для вывода элементов массива
cout << array1[counter] << " "; // выводим элементы массива на стандартное устройство вывода
for ( int counter = 0; counter < 10; counter++ ) // цикл для суммирования чисел массива
sum += array1[counter]; // суммируем элементы массива
cout << "}\nsum = " << sum << endl;
system("pause");
return 0;
}
```

**Двумерные массивы в С++**

В двумерном массиве есть две характеристики: количество строк и количество столбцов. Также и в двумерном массиве, кроме количества элементов массива, есть такие характеристики как, количество строк и количество столбцов двумерного массива. То есть, визуально, двумерный массив — это обычная таблица, со строками и столбцами. Фактически двумерный массив — это одномерный массив одномерных массивов. Структура двумерного массива, с именем a, размером m на n показана ниже.

где, m - количество строк двумерного массива; n - количество столбцов двумерного массива; m * n - количество элементов массива.1

В объявлении двумерного массива, также как и в объявлении одномерного массива, первым делом, нужно указать: тип данных; имя массива. После чего, в первых квадратных скобочках указывается количество строк двумерного массива, во вторых квадратных скобочках — количество столбцов двумерного массива. Двумерный массив визуально отличается от одномерного второй парой квадратных скобочек.

```
// пример объявление двумерного массива:
int a[5][3];
```

a - имя целочисленного массива, число в первых квадратных скобках указывает количество строк двумерного массива, в

данном случае их 5; число во вторых квадратных скобках указывает количество столбцов двумерного массива, в данном случае их 3.

```
// инициализация двумерного массива:
int a[5][3] = { {4, 7, 8}, {9, 66, -1}, {5, -5, 0}, {3, -3, 30}, {1, 1, 1} };
```

**Пример**

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
#include <iomanip>
using namespace std;
#define n 3
#define m 2
int main(int argc, char *argv[])
{
int mas[n][m];
srand(time(NULL));
for (int i=0; i<n; i++)
{
for (int j=0; j<m; j++)
{
mas[i][j]=30+rand()%21; // генерация числе в интервале от 30 до 50
cout<<setw(4)<<mas[i][j];
}
cout<<endl;
}
system("PAUSE");
return 0;
}
```

**Генерация случайных чисел в заданном диапазоне**

```
m = a + rand() % b;
```

где a - это начальная точка, с которой начинается генерация,

b - это величина сдвига, определяющая интервал, на котором будет производиться генерация.

### 3.2. Задание к лабораторной работе

**ЗАДАНИЕ 1.** Даны координаты точки на плоскости. Определить и вывести на экран номер квадранта, в который попадает точка.

**ЗАДАНИЕ 2.** Написать программу вычисления корней уравнения a*x*x+b*x+c=0. Значение корня квадратного от x возвращает функция стандартной библиотеки double sqrt(double x) (прототип в файле math.h).

**ЗАДАНИЕ 3.** Написать программу вычисления N! (использовать циклы for и do while).

**ЗАДАНИЕ 4.** Написать программу нахождения суммы квадратов всех нечетных чисел.

**ЗАДАНИЕ 5.** Найти количество элементов одномерного массива A(10), в значении которых установлен пятый бит. После этого у всех элементов массива инвертировать 3 бит и новые значения записать в массив B(10). Вывести в шестнадцатеричном виде массивы A и B.

**ЗАДАНИЕ 6.** Найти среднее арифметическое положительных элементов главной и побочной диагоналей матрицы действительных чисел A(5X5).

**ЗАДАНИЕ 7.** Дана матрица целых чисел размера 5x9. Получить одномерный массив, состоящий из средних арифметических элементов каждого из столбцов, имеющих четные номера. Найти максимальный элемент одномерного массива.

**ЗАДАНИЕ 8.** В реализации интерактивной игры необходимо отслеживать перемещение, в пределах игрового поля, объекта, управляемого игроком. Представим в программе игровое поле массивом целых значений Area(20X20). В начале все элементы имеют значение ноль. Положение перемещаемого объекта в конкретный момент времени можем задать, изменяя значение нужного элемента Area с нуля на единицу. После нескольких перемещений "след", оставляемый объектом будет выглядеть как цепочка единиц в поле нулей. Напишите собственную функцию int move(), которая, с помощью функции int getch() (прототип в файле conio.h), возвращающей код нажатой клавиши, определяет, в какую сторону сместился объект за один ход и возвращает 0, если сделан неправильный ход, -1, если сделан шаг влево, 1 - шаг вправо, -11 - вверх и 11 - вниз (можно использовать любые другие значения). Пользователь должен нажимать на клавиши с символами L или I, R или г, U или u, D ИЛИ d, соответственно. Используйте оператор switch. Напишите программу тест, создающую игровое поле, вы-зывающую некоторое число раз функцию move

и отображающую на экране результат.

### 3.3. Контрольные вопросы

1. Сколько операторов (управляющих конструкций) есть в Си/С++?

2. Для чего используется составной оператор (блок {})?

3. Можно ли в составном операторе размещать описания?

4. Какие из следующих конструкций синтаксически правильные:

```
if( -100 ) cout<<-100<<endl;
if( true ) cout<<true;
if( x > y ) z = x else z=y;
if( !x ) z = 1; else z = x;
if( x > 10 ) y = 1; z = x; else y = 0; z = -10;
```

5. Какие из следующих конструкций синтаксически правильные:

```
int i = 10; for(; !i; i--) cout<<i<<endl;
for(;;);
for(int i=0; i<10;) i++;
for(int j=0, j<10, j++) cout<<j<<endl;
for(int i=0,j=10; i!=j; ++i,--j)
cout<<"i="<<i<<"\tj="<<j<<endl;
```

6. Что отображается на экране в результате выполнения правильных конструкций

из вопроса 5?

7. Какие из следующих конструкций синтаксически правильные:

```
1) double d = 1.2;
switch(d) {
case 1.2: cout<<d<<endl;
}
2) char ch = 'A';
switch(ch) {
case 'A': case 'a':
cout<<'A'<<endl;
case 'B': case 'b':
cout<<'B'<<endl;
default:
cout<<'0'<<endl;
}
3) switch( ch ) {
case 65:
cout<<'A'<<endl;
```

```
break;
case 66:
cout<<'B'<<endl;
break;
}
```

8. Что отображается на экране в результате выполнения правильных конструкций
из вопроса 7?

9. В чем разница между оператором цикла с предусловием от опреатора цикла с
постусловием?

10. Какие из конструкций представляют оператор цикла с предусловием:

```
for(int i=0; i<10; ++i);
for(int i=0; ; ++i) if( i<10) break;
int i = 0; while( i<10 ) ++i;
int i= 0; while( 1 ) if( i == 10 ) break; else i++;
int i=0; do i++; while(i<10);
```

11. В чем разница между операторами break и continue?

12. Какие из следующих объявлений правильные:

```
int x[10], double y[20];
char str[] = { 'A', 'a', '\0' };
double d1[10], d2[5] = { 1, 2, 3};
float f[4] = { {1.1}, {1.2}, {1.3}, {1.4} };
char s[10] = "Hello";
```

13. Какие из следующих объявлений правильные:

```
double d[][] = { {1.0, 1.1, 1.2}, {1.3,1.4,1.5} };
double d[2][] = { {1.0, 1.1, 1.2}, {1.3,1.4, 1.5} };
double d[2][3] = { {1.0, 1.1, 1.2}, {1.3,1.4,1.5} };
double d[2][3] = { {1.0, 1.1, 1.2}, {1.3} };
double d[2][3] = { 1.0, 1.1, 1.2, 1.3,1.4, 1.5 };
```

14. Дан массив int x[3]. Какие из обращений к элементам массива правильные

```
x[1] = 1;
x[3] = x[1];
x[0] = x[1] = x[2];
x[1*2] = 1*2;
1[x] = 1;
```

15. Дан массив double y[2][3]. Какую матрицу можно представить с помощью
такого массива в программе

1.1 1.2

A = 1.3 1.4
1.5 1.6
1.1 1.2 1.3
B = 1.4 1.5 1.6
1 2
C = 3 4
5 6
1 2 3
D =

### 3.4. Литература

1. Brian W. Kernighan, Dennis M. Ritchie. "The C programming language". Second edition – Prentice Hall. – 1988, 272 p.

2. Стэнли Б. Липпман. "Язык программирования C++. Вводный курс". – М.: Вильямс. – 2007, 896 с.

3. Г. Шилдт. "Самоучитель C++". – СПб.: БХВ-Петербург. – 2001, 688 с.

4. К.Арнольд, Дж.Гослинг, Д.Холмс. "Язык программирования Java". 3-е изд. – М.: Вильямс. – 2001, 624 с.

## 4. ЛАБОРАТОРНАЯ РАБОТА №4: РАБОТА С МАССИВАМИ

### 4.1. Теория

#### I. Описание и объявление массивов в Си.

Массив – это несколько переменных одного типа, занимающих непрерывную область памяти. Доступ к конкретной переменной (элементу массива) производится с указанием имени массива и номера (индекса) элемента. Общий синтаксис описания одномерного массива таков:

тип_элементов *идентификатор*[константное целочисленное выражение];

Типом элементов массива не может быть пустой тип (void) и не может быть функция. Но может быть нетипизированный указатель (void *) и указатель на функцию. В следующем описании

int x[20], y;

x – это массив из 20-ти целочисленных переменных, а y просто переменная типа int.

Индексирование (нумерация) в массивах Си всегда начинается с нуля, поэтому, например, в массиве из трех переменных есть нулевой, первый и второй элементы:

int a[3];

```
a[0] = a[1] = a[2] = 1;
```

Исполняющая система Си (C Runtime) не контролирует выход значения индекса за допустимые пределы. Например,

```
int index = -1;
scanf ("%d", &index);
a[index] = 10;
```

приводит к выходу за допустимые для индекса элементов массива a значения (от нуля до двух), если пользователь вводит неподходящее значение. Такую ошибку нельзя обнаружить на этапе компиляции программы. Она проявляется только во время ее выполнения.

Многомерные массивы в Си – это одномерные массивы, состоящие из сложных элементов (массивов). Например, математическое понятие "матрица" может быть отображено на двумерный массив, в котором первая размерность задает число строк матрицы, а вторая – число элементов в каждой строке, т.е. число столбцов матрицы. Например,

```
double matrix[3][2];
```

представляет матрицу из трех строк и дух столбцов, содержащую вещественные числа.

## II. Инициализация массивов.

При объявлении массива можно задать начальные значения для его элементов – проинициализировать массив. Например,

```
char ch1[5] = {'H', 'e', 'l', 'l', 'o'};
int arr1[10] = { 1, 2, 3 }, arr2[20] = {};
double d[] = { 1.1, 1.2, 2., 3.5 };
char ch2[] = "Hello";
```

Для массива ch1 все значения для элементов массива заданы явно, в массиве arr1 первые три элемента (с индексами 0, 1 и 2) получают, соответственно, значения 1, 2 и 3. Оставшиеся семь элементов получают нулевые значения. В массиве arr2 значения по умолчанию (нулевые) получают все двадцать элементов. Для массивов d и ch2 размер задан неявно. Для массива d задано четыре инициализирующих выражения, поэтому компилятор выделит под массив блок памяти, нужный для размещения 4 переменных типа double (32 байта). Подобным же образом под массив ch2 будет выделено шесть байт (с учетом символа с кодом ноль – терминатора строки).

Многомерные массивы могут инициализироваться так же, как и одномерные, или с помощью специального синтаксиса с вложенными скобками:

```
double mat1[3][2] = { 1., 2., 3., 4., 5., 6. };
double mat2[3][2] = {
    { 1., 2. },
    { 3., 4. },
    { 5., 6. }
};
```

### III. Передача массивов в функции.

Если массив является аргументом функции, то его элементы не копируются в программный стек. Передается (копируется в стек) адрес нулевого элемента массива. Поэтому, если, например, объявлена функция

```
void f(int a[10]);
```

значение 10 не будет передаваться функции. Компилятор просто игнорирует размер массива. Тем самым в функцию f можно передавать одномерные массивы разных размеров, если тип элементов этих массивов int. Чтобы функция знала, какого размера массив ей передан, нужно использовать еще один аргумент, например

```
void f(int a[], int size);
```

## 4.2. Задание к лабораторной работе

1. В стандартной библиотеке реализована функция rand(), возвращающая псевдослучайные числа из диапазона [0.0, 1.0) с равномерным распределением. Прототип функции размещен в файле заголовков <math.h>. Создайте одномерный массив из 100 действительных чисел. Заполните его случайными значениями из диапазона [-10, 10). Посчитатйте, сколько раз в массиве встретилось значение 0.0.

2. Напишите собственную функцию, определяющую среднее арифметическое отрицательных элементов переданного ей одномерного массива.

3. В стандартной библиотеке есть функция pow(double x, double y), вычисляющая значение $x^y$ (только для положительных значений y). Напишите собственную функцию, определяющую среднее геометрическое положительных элементов переданного ей одномерного массива.

4. Напишите программу для нахождения суммы элементов главной и побочной диагоналей матрицы размером 3x3. Значения элементов матрицы должны задаваться с помощью конструкции инициализации.

5. Напишите программу, выполняющую транспонирование матрицы размером 4x5.

### 4.3. Контрольные вопросы

1. Что такое массив?
2. Какие типы недопустимы для элементов массива?
3. Каков индекс первого элемента массива?
4. Дан массив int a[10]; Какие из следующих выражений правильные

a[10] = 1
a[0] = 1.1
a[1] = -1
a[-1] = 1
2[a] = 2

5. Как можно представить матрицу в программе?
6. Сколько строк и столбцов в матрице, представленной в программе массивом double a[3][4]?
7. Что возвращает операция индексирования?
8. Как проинициализировать одномерный массив?
9. Как проинициализировать многомерный массив?
10. Что такое неполная инициализация?
11. Как неявно задать размер массива при его объявлении?

### 4.4. Литература

1. Brian W. Kernighan, Dennis M. Ritchie. "The C programming language". Second edition – Prentice Hall. – 1988, 272 p.
2. Стэнли Б. Липпман. "Язык программирования C++. Вводный курс". – М.: Вильямс. – 2007, 896 с.
3. Г. Шилдт. "Самоучитель C++". – СПб.: БХВ-Петербург. – 2001, 688 с.

## 5. ЛАБОРАТОРНАЯ РАБОТА №5: УКАЗАТЕЛИ В C/C++

### 5.1. Теория

**Указатель** - это переменная, предназначенная для хранения адреса некоторого объекта. Различают типизированные и не типизированные указатели (void *), указатели на данные и на функции.

Множества допустимых операций для указателей разных видов различны. Так, для не типизированных указателей запрещены операции разадресации, инкремента и т.п. Для указателей на функции (но не на данные) определена операция вызова функции.

**Указатели используются для передачи по ссылке данных,** что намного ускоряет процесс обработки этих данных (в том случае, если объѐм данных большой), так как их не надо копировать, как при передаче по значению, то есть, используя имя переменной.

В основном указатели **используются для организации динамического распределения памяти**, например при объявлении массива, не надо будет его ограничивать в размере. Ведь программист заранее не может знать, какого размера нужен массив тому или иному пользователю, в таком случае используется динамическое выделение памяти под массив.

Любой указатель необходимо **объявить** перед использованием, как и любую переменную. Принцип объявления указателей такой же, как и принцип объявления переменных. Отличие заключается только в том, что перед именем ставится символ звѐздочки *.

**Пример**

```
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
int var = 123; // инициализация переменной var числом 123
int *ptrvar = &var; // указатель на переменную var (присвоили адрес переменной указателю)
cout << "&var = " << &var << endl;// адрес переменной var содержащийся в памяти, извлечѐнный операцией взятия адреса
cout << "ptrvar = " << ptrvar << endl;// адрес переменной var, является значением указателя ptrvar
cout << "var = " << var << endl; // значение в переменной var
cout << "*ptrvar = " << *ptrvar << endl; // вывод значения содержащегося в переменной var через указатель, операцией разименования указателя
system("pause");
return 0;
}
```

**Указатели можно сравнивать**, причѐм не, только на равенство или неравенство, ведь адреса могут быть меньше или больше относительно друг друга.

**Пример**

```
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
int var1 = 123; // инициализация переменной var1 числом 123
int var2 = 99; // инициализация переменной var2 числом 99
int *ptrvar1 = &var1; // указатель на переменную var1
int *ptrvar2 = &var2; // указатель на переменную var2
cout << "var1 = " << var1 << endl;
cout << "var2 = " << var2 << endl;
cout << "ptrvar1 = " << ptrvar1 << endl;
cout << "ptrvar2 = " << ptrvar2 << endl;
if (ptrvar1 > ptrvar2) // сравниваем значения указателей, то есть адреса переменных
cout << "ptrvar1 > ptrvar2" << endl;
if (*ptrvar1 > *ptrvar2) // сравниваем значения переменных, на которые ссылаются указатели
cout << "*ptrvar1 > *ptrvar2" << endl;
system("pause");
return 0;
}
```

Указатель на объект любого типа можно присвоить переменной типа void*.

void* можно присваивать, сравнивать и явно преобразовать в указатель любого другого типа.

**Пример**

```
#include <stdio.h>
void main (void)
{
int a =123;
double d= 3.45678;
// объвление указателя на неопределенный тип
void *vp;
//указатель на неопределенный тип инициализирован адресом объекта целого типа
vp=&a;
//перед выводом значения объекта целого типа, адресуемого указателем vp,
// тип указателя явно приводится к целому
printf("a=%d \n", *((int *)vp)); /*указатель на неопределенный тип инициализирован адресом объекта типа double */
```

```
vp=&d;
//перед выводом значения объекта типа double , адресуемого указателем vp,
// тип указателя явно приводится к типу double
printf("d=%lf \n", *((double *)vp));
}
```

**Указатели могут ссылаться на другие указатели.** При этом в ячейках памяти, на которые будут ссылаться первые указатели, будут содержаться не значения, а адреса вторых указателей. Число символов * при объявлении указателя показывает порядок указателя. Чтобы получить доступ к значению, на которое ссылается указатель его необходимо разыменовывать соответствующее количество раз. Разработаем программу, которая будет выполнять некоторые операции с указателями порядка выше первого.

**Пример**

```
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
int var = 123; // инициализация переменной var числом 123
int *ptrvar = &var; // указатель на переменную var
int **ptr_ptrvar = &ptrvar; // указатель на переменную var
int ***ptr_ptr_ptrvar = &ptr_ptrvar;
cout << " var\t\t= " << var << endl;
cout << " *ptrvar\t= " << *ptrvar << endl;
cout << " **ptr_ptrvar = " << **ptr_ptrvar << endl; // два раза разименовываем указатель, так как он второго порядка
cout << " ***ptr_ptrvar = " << ***ptr_ptr_ptrvar << endl; // указатель третьего порядка
cout << "\n ***ptr_ptr_ptrvar -> **ptr_ptrvar -> *ptrvar -> var -> "<< var << endl;
cout << "\t " << &ptr_ptr_ptrvar<< " -> " << " " << &ptr_ptrvar << " ->" << &ptrvar << " -> " << &var << " -> " << var << endl;
system("pause");
return 0;
}
```

Указатели позволяют максимально эффективно решить проблему возврата нескольких значений из функции. В случае, если требуется вернуть несколько значений, можно воспользоваться следующим синтаксисом:

```
void GetMouseState(int *p_keys, int *p_x, int *p_y) {
*p_keys = …;
*p_x = …;
*p_y = …;
}
```

Пояснение: поскольку функция GetMouseState получает копии адресов объектов, то при изменении их через указатели меняются сами объекты, а не их копии.

Вызов этой функции будет выглядеть следующим образом:

```
int keys,x,y;
GetMouseState(&keys,&x,&y);
```

**Имя массива можно рассматривать как указатель на его первый элемент**.

**Пример**

```
int v[] = { 1,2,3,4 };
int *p1 = v; // Указатель на первый элемент
int *p2 = &v[0] // Указатель на первый элемент
int *p3 = &v[4] // Указатель на элемент, следующий за последним
```

При передаче массива как аргумента функции происходит неявное преобразование имени массива в указатель на его начальный элемент с потерей информации о размере массива. Таким образом, массив всегда передаѐтся по указателю – его копия не создаѐтся

Определение **массива указателей** выглядит следующим образом:

int *ap[15]; // Массив из 15 указателей на int

Результат применения операторов -, +, --, ++ к указателю зависит от типа объекта, на который ссылается указатель. Если к указателю p типа T* применяется арифметическая операция, предполагается, что он указывает на элемент массива объектов типа T; p+1 указывает на следующий элемент массива, а p-1 на предыдущий. То есть целое значение p+1 будет на sizeof(T) больше, чем целое значение p.

**Пример**

```
int arr[ArraySize];
for (int i=0; i < ArraySize; ++i)
arr[i] = 0;
```

с использованием указателей:

```
int arr[ArraySize];
int *p=arr;
```

```
for (int i=0; i < ArraySize; ++i)
*p++ = 0;
```

Пояснение: унарные операторы * и ++ имеют одинаковый приоритет, однако они правоассоциативны. То есть в данном случае первым будет выполняться оператор ++, увеличивающий значение указателя. Указатель будет сдвинут на следующий элемент массива. Но поскольку это оператор постинкремента, то для разыменования будет использовано старое значение указателя. Таким образом, в одном выражении записано сразу два действия: передвинуть указатель и разыменовать указатель. В объект, на который указывает указатель помещается ноль.

**Пример**

Передача строки в функцию

```
void show_string(char *string)
{
while (*string != '\0')
{
cout << *string;
string++;
}
}
```

Аналогичная программа, но в более компактной записи

```
while (*string) { cout << *string++; }
```

**Пример**

Передача числового массива в функцию

```
void show_float(float *array, int number_of_elements)
{
int i;
for (i = 0; i < number_of_elements; i++) cout << *array++
<< endl;
}
void main(void)
{
float values[5] = {1.1, 2.2, 3.3, 4.4, 5.5);
show_float(values, 5);
}
```

**Указатели могут ссылаться на функции.** Имя функции, как и имя массива само по себе является указателем, то есть содержит адрес входа.

**Пример**

```
#include <iostream>
```

```
using namespace std;
int nod(int, int ); // прототип указываемой функции
int main(int argc, char* argv[])
{
int (*ptrnod)(int, int); // объявление указателя на функцию
ptrnod=nod; // присваиваем адрес функции указателю ptr-
nod
int a, b;
cout << "Enter first number: ";
cin >> a;
cout << "Enter second number: ";
cin >> b;
cout << "NOD = " << ptrnod(a, b) << endl; // обращаемся к
функции через указатель
system("pause");
return 0;
}
int nod(int number1, int number2) // рекурсивная функция
нахождения наибольшего общего делителя НОД
{
if ( number2 == 0 ) //базовое решение
return number1;
return nod(number2, number1 % number2); // рекурсивное
решение НОД
}
```

**Выделение и освобождение памяти**

**Си**

**void* malloc(size_t); //функция выделения памяти**

**void free(void* memblock); //функция освобождения памяти**

Здесь size_t – размер выделяемой области памяти в байтах; void* - обобщенный тип указателя, т.е. не привязанный к какому-либо конкретному типу.

**Пример**

```
#include <stdlib.h>
int main()
{
double* ptd;
ptd = (double *)malloc(10 * sizeof(double)); // выделения па-
мяти для 10 элементов
//типа double.
if(ptd != NULL)
```

```
{
for(int i = 0;i < 10;i++)
ptd[i] = i;
} else printf("Не удалось выделить память.");
free(ptd);
return 0;
}
```

**void *calloc(size_t num, size_t size);**

Функция calloc() выделяет память, размер которой равен значению выражения num * size, т.е. память, достаточную для размещения массива, содержащего num объектов размером size. Все биты распределенной памяти инициализируются нулями.

Функция calloc() возвращает указатель на первый байт выделенной области памяти. Если для удовлетворения запроса нет достаточного объема памяти, возвращается нулевой указатель. Перед попыткой использовать распределенную память важно проверить, что возвращаемое значение не равно нулю.

**Пример**

Эта функция возвращает указатель на динамически распределенный блок памяти для массива из 100 чисел типа float:

```
#include <stdlib.h>
#include <stdio.h>
float *get_mem(void)
{
float *p;
p = calloc(100, sizeof(float));
if(!p) {
printf("Ошибка при распределении памяти\n");
exit(1);
}
return p;
}
```

**void *realloc(void *ptr, size_t size);**

Функция realloc() изменяет размер блока ранее выделенной памяти, адресуемой указателем ptr в соответствии с заданным размером size. Значение параметра size может быть больше или меньше, чем перераспределяемая область. Функция realloc() возвращает указатель на блок памяти.

**Пример**

Эта программа сначала выделяет блок памяти для 17 символов, а затем использует realloc() для увеличения размера блока до 18 символов, чтобы разместить в конце точку.

```
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
char *p;
p = malloc(17);
if(!p) {
printf("Ошибка при распределении памяти\n");
exit(1);
}
p = realloc(p, 18);
if(!p) {
printf("Ошибка при распределении памяти\n");
exit(1);
}
printf(p);
free(p);
return 0;
}
```

**C++**

Операции **new и delete** предназначены для динамического распределения памяти компьютера. Операция **new** выделяет память из области свободной памяти, а операция **delete** высвобождает выделенную память.

Выделяемая память, после еѐ использования должна высвобождаться, поэтому операции **new и delete** используются парами. Даже если не высвобождать память явно, то она освободится ресурсами ОС по завершению работы программы.

**Пример**

```
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
int *ptrvalue = new int; // динамическое выделение памяти
под объект типа int
*ptrvalue = 9; // инициализация объекта через указатель
//int *ptrvalue = new int (9); инициализация может выпол-
```

нятся сразу при объявлении динамического объекта

```
cout << "ptrvalue = " << *ptrvalue << endl;
delete ptrvalue; // высвобождение памяти
system("pause");
return 0;
}
```

**Создание динамического массива**

// объявление одномерного динамического массива на 10 элементов:

**float *ptrarray = new float [10];**

// где ptrarray – указатель на выделенный участок памяти под массив вещественных чисел типа float

// в квадратных скобочках указываем размер массива

// высвобождение памяти отводимой под одномерный динамический массив:

**delete [] ptrarray;**

**Пример**

```
#include <iostream>
// в заголовочном файле <ctime> содержится прототип функции time()
#include <ctime>
// в заголовочном файле <iomanip> содержится прототип функции setprecision()
#include <iomanip>
using namespace std;
int main(int argc, char* argv[])
{
srand(time(0)); // генерация случайных чисел
float *ptrarray = new float[10]; // создание динамического массива вещественных чисел на десять элементов
for (int count = 0; count < 10; count++)
ptrarray[count] = (rand() % 10 + 1) / float((rand() % 10 + 1)); //заполнение массива случайными числами с масштабированием от 1 до 10
cout << "array = ";
for (int count = 0; count < 10; count++)
cout << setprecision(2) << ptrarray[count] << " ";
delete [] ptrarray; // высвобождение памяти
cout << endl;
system("pause");
```

```
return 0;
}
```

**Двумерный динамический массив**

// объявление двумерного динамического массива на 10 элементов:

```
float **ptrarray = new float* [2]; // две строки в массиве
for (int count = 0; count < 2; count++)
ptrarray[count] = new float [5]; // и пять столбцов
```

// где ptrarray – массив указателей на выделенный участок памяти под массив вещественных чисел типа float

**// высвобождение памяти отводимой под двумерный динамический массив:**

```
for (int count = 0; count < 2; count++)
delete [] ptrarray[count];
// где 2 – количество строк в массиве
```

**Пример**

```
#include <iostream>
#include <ctime>
#include <iomanip>
using namespace std;
int main(int argc, char* argv[])
{
srand(time(0)); // генерация случайных чисел
// динамическое создание двумерного массива вещественных чисел на десять элементов
float **ptrarray = new float* [2]; // две строки в массиве
for (int count = 0; count < 2; count++)
ptrarray[count] = new float [5]; // и пять столбцов
// заполнение массива
for (int count_row = 0; count_row < 2; count_row++)
for (int count_column = 0; count_column < 5; count_column++)
ptrarray[count_row][count_column] = (rand() % 10 + 1) / float((rand() % 10 + 1)); //заполнение массива случайными числами с масштабированием от 1 до 10
// вывод массива
for (int count_row = 0; count_row < 2; count_row++)
{
for (int count_column = 0; count_column < 5; count_column++)
```

```
cout << setw(4) <<setprecision(2) << ptrarray[count_row][count_column] << " ";
cout << endl;
}
// удаление двумерного динамического массива
for (int count = 0; count < 2; count++)
delete []ptrarray[count];
system("pause");
return 0; }
```

## 5.2. Задание к лабораторной работе

**ЗАДАНИЕ 1.** В программе объявлен массив:
int P [ ]={0, 2, 4, 5, 6, 7, 9, 12};
Какие значения примут выражения:
а) P [3]; б) *P; в) *(P+4); г) *(P+P[2])?

**ЗАДАНИЕ 2.** Написать функцию, которая меняет местами значения двух переменных x и y.

**ЗАДАНИЕ 3.** Для решения различных задач используются методы Монте-Карло, предполагающие применение массивов случайных чисел с большим количеством элементов. Размер массива становится известным во время выполнения про-граммы, т.е. массив должен создаваться динамически. Создайте две функции для решения одной и той же задачи: динамическое создание и заполнение случайными числами массива указанного размера. Первая функция должна использовать возвращаемое значение для передачи пользователю сгенерированного массива, а вторая должна передавать массив через один из своих аргументов. Стандартная библиотека Си содержит функции int rand() и void srand(unsigned) для генерации псевдослучайных чисел (прототипы в файле stdlib.h).

**ЗАДАНИЕ 4.** Дана строка, содержащая текст на естественном языке. Напишите функцию, создающую новую строку, в которой все слова из старой строки следуют в обратном порядке и разделены одним знаком пробела. Исходная строка может содержать различные знаки-разделители (пробелы, запятые, точки и т.п.). Полный набор знаков-разделителей передается функции при ее вызове.

**ЗАДАНИЕ 5.** Напишите универсальную функцию для

нахождения среднего гео-метрического отрицательных элементов матриц с произвольным числом строк и столбцов. Напишите программу-тест с промежуточной конструкцией, позволяющей передавать в функцию двумерные массивы.

**ЗАДАНИЕ 6.** Напишите функцию, строящую график заданной функции на заданном интервале изменения аргумента. Указатель на конкретную функцию и предельные значения аргумента передаются через аргументы. Алгоритм работы функции может быть таким: разбить интервал изменения аргумента на фиксированное число равновеликих подинтервалов и для каждого значения аргумента найти значение функции. Получим сетку значений аргумента и функции. Затем нужно провести масштабирование - перейти от действительных значений к целым - экранным координатам, полагая, что ось абсцисс направлена сверху-вниз, а ось ординат - слева-направо (график "лежит на боку"). Точки, соответствующие значениям функции на сетке можно отображать с помощью какого-нибудь символа, например, звездочки. Схема масштабирования:

double x, y; double yMin, yMax; /* минимальное и максимальное значения функции на
заданном интервале изменения x */
const int yScrMin = 1, yScrMax = 50; /* пределы изменения "экранной" координаты
yScr */
int yScr; /* координата на экране, соответствующая "физической" координате y */
/* найти yMin и YMax */
У = f(x);
/* масштабирование */
yScr = yScrMin + (y - yMin)/(yMax - yMin)*(yScrMax - yScrMin);

Рисунок для функции y = x * x

```
|-----------------------------------------------------------
| *
| *
| *
| *
| *
| *
| *
```

**ЗАДАНИЕ 7.** Создайте текстовый, основанный на использовании меню, интер-фейс пользователя для тестирования функций, содержащих алгоритмы решения заданий 1-4. Используйте массивы указателей.

## 5.3. Контрольные вопросы

1. Что такое указатель?
2. Как объявить указатель?
3. Что такое *void?
4. Что будет выводиться на экран?

```
int i=123 *pi1=&i, *pi2;
pi2=pi1;
printf(“ %p %p”, pi1, pi2);
```

5. Чему равен y?

```
int x=5, y, *px=&x;
y = *px + 5;
```

6. Как работает операция инкремента и декремента для указателя?
7. Что будет выводиться на экран?

```
#include <stdio.h>
void main (void)
{
int x=5,y,*px=&x;
y=*px+5;
printf("y=%d значение указателя=%p\n",y,px);
y=*px++;
printf("y=%d значение указателя=%p\n",y,px);
px=&x; y=(*px)++;
printf("y=%d значение указателя=%p значение, адресуемое указателем *px= %d\n",y,px,*px);
```

8. Что будет выводиться на экран?

```
char s[]="hello";
char *p=&s[3];
cout << *p << endl;
p++;
cout << *p << endl;
```

9. Что будет выводиться на экран?

```
int i[]={1,2,3,4};
int *pI=&i[2];
cout << *pI << endl; pI++;
cout << *pI << endl;
```

10. Чем отличаются функции malloc и calloc?
11. Можно ли с помощью функции realloc уменьшить размер захваченного прежде блока памяти?
12. Можно ли освободить часть захваченного прежде с помощью malloc или calloc блока памяти?
13. В чем преимущества и недостатки операций new/delete по сравнению с функциями динамического распределения памяти Си?
14. Для чего можно использовать указатель на указатель?
15. Можно ли освободить блок памяти, захваченный с помощью операции new функцией free?

### 5.4. Литература

1. Brian W. Kernighan, Dennis M. Ritchie. "The C programming language". Second edition – Prentice Hall. – 1988, 272 p.
2. Стэнли Б. Липпман. "Язык программирования C++. Вводный курс". – М.: Вильямс. – 2007, 896 с.
3. Г. Шилдт. "Самоучитель C++". – СПб.: БХВ-Петербург. – 2001, 688 с.
4. К.Арнольд, Дж.Гослинг, Д.Холмс. "Язык программирования Java". 3-е изд. – М.: Вильямс. – 2001, 624 с.

## 6. ЛАБОРАТОРНАЯ РАБОТА №6: ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

### 6.1. Теория

Узел списка представляется в программе структурой, поля которой отражают его содержательную и управляющую части. Содержательная часть определяется особенностями решаемой задачи, а управляющая часть – это указатели на предыдущий и следующий узлы. Представлением для списка в целом является указатель на первый элемент списка. Для пустого списка – пустой указатель (NULL).

**Выделяются следующие динамические структуры данных:**

однонаправленные (односвязные) списки;

двунаправленные (двусвязные) списки;
циклические списки;
стек;
дек;
очередь;
бинарные деревья.

**Объявление динамических структур данных**

Каждая компонента любой динамической структуры представляет собой запись, содержащую, по крайней мере, два поля: одно поле типа указатель, а второе – для размещения данных. В общем случае запись может содержать не один, а несколько указателей и несколько полей данных. Поле данных может быть переменной, массивом или структурой. Для наилучшего представления изобразим отдельную компоненту в виде:

| P |
|---|
| D |

где поле P – указатель; поле D – данные.

**Элемент динамической структуры состоит из двух полей:**

информационного поля (поля данных), в котором содержатся те данные, ради которых и создается структура; в общем случае информационное поле само является интегрированной структурой – вектором, массивом, другой динамической структурой и т.п.;

адресного поля (поля связок), в котором содержатся один или несколько указателей, связывающий данный элемент с другими элементами структуры;

Объявление элемента динамической структуры данных выглядит следующим образом:

**struct имя_типа {**
**информационное поле;**
**адресное поле;**
**};**

**Например:**

```
struct TNode {
int Data; //информационное поле
TNode *Next; //адресное поле
};
```

Информационных и адресных полей может быть как одно, так и несколько.

Рассмотрим в качестве примера динамическую структуру, схематично указанную на рис.

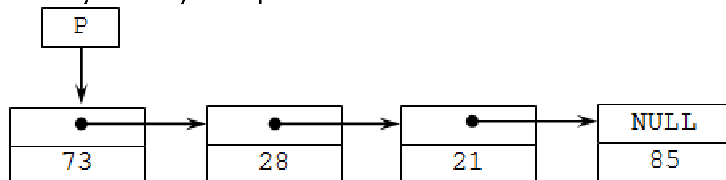


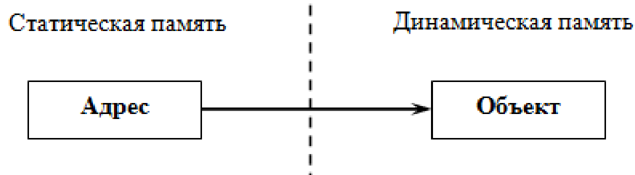

Рис. Схематичное представление динамической структуры

Данная структура состоит из 4 элементов. Ее первый элемент имеет поле Data, равное 73, и связан с помощью своего поля Next со вторым элементом, поле Data которого равно 28, и так далее до последнего, четвертого элемента, поле Data которого равно 85, а поле Next равно NULL, то есть нулевому адресу, что является признаком завершения структуры. Здесь P является указателем, который указывает на первый элемент структуры.

**Доступ к данным в динамических структурах**

Элемент динамической структуры в каждый момент может либо существовать, либо отсутствовать в памяти, поэтому его называют динамическим. Поскольку элементами динамической структуры являются динамические переменные, то единственным средством доступа к динамическим структурам и их элементам является указатель (адрес) на место их текущего расположения в памяти. Таким образом, доступ к динамическим данным выполняется специальным образом с помощью указателей.

Указатель содержит адрес определенного объекта в динамической памяти. Адрес формируется из двух слов: адрес сегмента и смещение. Сам указатель является статическим объектом и расположен в сегменте данных.




Для обращения к динамической структуре достаточно хранить в памяти адрес первого элемента структуры. Поскольку каждый элемент динамической структуры хранит адрес следующего за ним элемента, можно, двигаясь от начального элемента по ад-

ресам, получить доступ к любому элементу данной структуры.

Доступ к данным в динамических структурах осуществляется с помощью операции "стрелка" ( -> ), которую называют операцией косвенного выбора элемента структурного объекта, адресуемого указателем. Она обеспечивает доступ к элементу структуры через адресующий ее указатель того же структурного типа.

Формат применения данной операции следующий:

**УказательНаСтруктуру-> ИмяЭлемента**

Операции "стрелка" ( -> ) двуместная. Применяется для доступа к элементу, задаваемому правым операндом, той структуры, которую адресует левый операнд. В качестве левого операнда должен быть указатель на структуру, а в качестве правого – имя элемента этой структуры.

**Например:**

```
p->Data;
p->Next;
```

**Работа с памятью при использовании динамических структур**

В программах, в которых необходимо использовать динамические структуры данных, работа с памятью происходит стандартным образом.

Выделение динамической памяти производится с помощью операции **new** или с помощью библиотечной функции **malloc (calloc).**

Освобождение динамической памяти осуществляется операцией **delete** или функцией **free**.

**Пример**

```
struct Node {char *Name;
int Value;
Node *Next
};
Node *PNode; //объявляется указатель
PNode = new Node; //выделяется память
PNode->Name = "STO"; //присваиваются значения
PNode->Value = 28;
PNode->Next = NULL;
delete PNode; // освобождение памяти
```

**Примеры На Паскале**

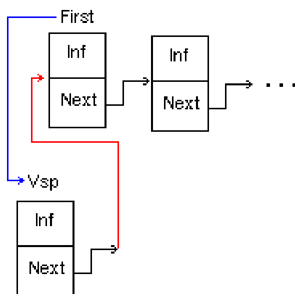
```
Type U = ^Zveno;
Zveno = Record
Inf : BT;
Next: U
End;
```

**На Си**

```
struct Zveno {
BT Inf;
Zveno * Next
};
```

**Здесь BT — некоторый базовый тип элементов списка.**

**1. Добавление звена в начало списка**



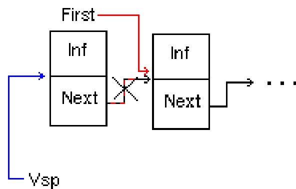

**На Паскале**

```
Procedure V_Nachalo(Var First : U; X : BT);
Var Vsp : U;
Begin
New(Vsp);
Vsp^.Inf := X;
Vsp^.Next := First; {То звено, что было заглавным, становится вторым по счѐту}
First := Vsp; {Новое звено становится заглавным}
End;
```

**На Си**

```
Zveno* Nachalo(Zveno* First, BT X)
{
Zveno* Vsp = new Zveno;
Vsp -> Inf = X;
Vsp->Next=First;
First=Vsp;
return First;
}
```

**2. Удаление звена из начала списка**




**На Паскале**

```
Procedure Iz_Nachala(Var First : U; Var X : BT);
Var Vsp : U;
Begin
Vsp := First; {Забираем ссылку на текущее заглавное звено}
First := First^.Next; {То звено, что было вторым по счёту, становится заглавным}
X := Vsp^.Inf; {Забираем информацию из удаляемого звена}
Dispose(Vsp); {Уничтожаем звено}
End;
```

**На Си**

```
Zveno* Iz_Nachala(Zveno* First, BT X)
{
Zveno* Vsp;
Vsp = First;
First = First->Next;
X = Vsp->Inf;
delete Vsp;
return First;
}
```

## 6.2. Задание к лабораторной работе

1. Разработайте динамическую структуру «**стек**» для решения задачи. Последовательность чисел Фибоначчи задается по закономерности: f1 = 1, f2 = 1, ..., fn = fn-1 + fn-2. Распечатайте n чисел Фибоначчи в следующем порядке: сначала все четные, затем все нечетные элементы. Все получаемые числа Фибоначчи хранятся в стеке.

2. Создайте функции и программу-тест, реализующие основные операции для очереди, представленной двунаправленным списком. Узлы списка содержат целочисленные значения.

3. Реализовать представление и основные операции для множеств. Помимо добавления, удаления, получения (и т.д.) элемента множества, реализовать операции объединения и пересе-

чения множеств.

4. Организовать линейный список цветов: хранить название цвета и его числовой код. Упорядочивать по первой букве названия, если первые буквы совпадают, то по второй и т.д. При добавлении нового цвета в список сохранять упорядоченность.

### 6.3. Контрольные вопросы

1. Чем отличаются статические и динамические величины?
2. Почему для обращения к динамической структуре достаточно хранить в памяти адрес ее первого элемента?
3. За счет чего работа с динамическими данными замедляет выполнение программы?
4. Какого типа может быть поле данных в динамической структуре?
5. Как в программе представить узел списка?
6. Как представить список в целом? Пустой список?
7. Как через указатель на узел получить доступ к содержимому узла?
8. Какую дисциплину добавления/удаления элементов поддерживает динамическая структура "стек"?
9. Какую дисциплину добавления/удаления элементов поддерживает динамическая структура "очередь"?
10. Удобно ли реализовать динамическую структуру "стек" ("очередь") на основе массива? Списка? Двоичного дерева?
11. Даны два множества {-1, 0, 1} и { 1, 2, 3}. Каким будет их пересечение? Объединение?
12. Сколько аргументов (и какого типа) должно быть у функции, реализующей алгоритм нахождения пересечения двух множеств, если тип возвращаемого ею значения void?
13. Элементы множества хранятся в упорядоченном виде. Какой алгоритм поиска элемента множества Вы можете предложить?
14. Какие динамические структуры данных, кроме указанных в этой лабораторной работе, Вы знаете?

### 6.4. Литература

1. Brian W. Kernighan, Dennis M. Ritchie. "The C programming language". Second edition – Prentice Hall. – 1988, 272 p.
2. Стэнли Б. Липпман. "Язык программирования C++. Вводный курс". – М.: Вильямс. – 2007, 896 с.
3. Г. Шилдт. "Самоучитель C++". – СПб.: БХВ-Петербург. – 2001, 688 с.

4. К.Арнольд, Дж.Гослинг, Д.Холмс. "Язык программирования Java". 3-е изд. – М.: Вильямс. – 2001, 624 с.

# 7. ЛАБОРАТОРНАЯ РАБОТА №7: ФАЙЛОВЫЙ ВВОД И ВЫВОД

## 7.1. Теория

### I. Понятие файлового потока

Файловый поток – это абстракция, отделяющая программу от конкретного устройства или файла. При создании поток связывается с конкретным устройством или файлом и, далее, в операциях чтения или записи используется поток. Таким образом, чтение и запись в программе не требуют явного указания устройства или файла и позволяют программисту не думать об особенностях соответствующих внешних устройств или файлов при работе с ними. При создании потока происходит обращение к операционной системе с запросом на выделение ресурса и, обычно, операционная система возвращает дескриптор (целое число), сохраняемое потоком на протяжении всего его использования. Кроме того, операции чтения/записи для медленных внешних устройств обычно кэшируются, т.е. в памяти создается буфер, и все операции чтения или записи обращаются к этому буферу. Только когда из буфера считана вся информация, или, при записи, буфер переполняется, происходит реальное обращение к внешнему устройству. Для потока определено также понятие указателя текущей позиции. При каждой операции чтения или записи указатель текущей позиции в файле смещается на прочитанное или записанное количество байт.

### II. Файловые потоки в Си

В Си файловый поток ввода/вывода представлен указателем на структуру FILE. Определение типа FILE, прототипы функций для работы с файловыми потоками содержатся в файле заголовков **stdio.h.**

Чтобы создать поток и связать его с файлом или устройством можно использовать функцию

```
FILE *fopen(char *name, char *mode);
```

name задает спецификацию файла или имя устройства, mode – режим работы с потоком. Возможны следующие режимы: r – чтение, w – запись и a – дозапись в конец файла, кроме того, можно использовать расширенные режимы r+ - чтение и запись,

w+ - запись и чтение, a+ - дозапись и чтение. Файл можно трактовать как текстовый или двоичный, t или b, соответственно. По умолчанию файл считается текстовым.

**Для текстового файла выделяются следующие режимы:**

"rt" - текстовый для чтения,

"wt" - текстовый для записи,

"at" - текстовый для дозаписи в уже существующий набор данных

**Для двоичного файла**

"rb" - двоичный для чтения,

"rb+" - двоичный для чтения и записи,

"wb" - двоичный для записи,

"wb+" - двоичный для записи и чтения.

Например,

```
FILE *f1, *f2, *f3;
f1 = fopen("a.txt","r"); // или f1 = fopen("a.txt","rt");
f2 = fopen("b.txt","wb");
f3 = fopen("c.txt","r+b");
```

f1 открывается для чтения в текстовом режиме, f2 – для записи в двоичном режиме, f3 – для чтения и записи в двоичном режиме.

Имя файла можно задавать относительно текущего каталога

```
FILE * fo;
fo = fopen("test.txt","wt");
```

или с указанием полного пути к файлу, например:

```
fo = fopen("c:\\tmp\\test.txt","wt");
```

Проверка: открыт файл или нет

```
if( (fo=fopen("c:\\tmp\\test.txt","rt")) == 0 ) {
// ошибка!
}
```

Разорвать связь потока с файлом и вернуть дескриптор файла операционной системе можно с помощью функции

void fclose(FILE *);

После вызова fclose файловую переменную уже нельзя использовать до той поры, пока она вновь не будет связана с потоком, возвращенным в результате нового обращения к функции fopen.

Запись текстовой строки в файл выполняет функция **fprintf():**

fprintf( имя-файловой-переменной, формат, список-переменных-для-вывода );

Формат - это текстовая строка, задающая формат записываемого в файл текста, так же, как и при использовании функции **printf**.

```
fprintf( fo, "Привет!" );
```

**Пример**

```
int n = 10;
char str[20] = "значение переменной n равно ";
fprintf( fo, "Вывод: %s %d", str, n );
```

**Пример**

```
FILE * fo;
fo = fopen("test.txt","wt");
int i;
for( i=0; i<100; i++ ) {
fprintf( fo, "%d\n", i );
}
fclose(fo);
```

Для чтения файла используется **fscanf().**

fscanf( файловая-переменная, формат-ввода, список-адресов-переменных )

**Например,** чтение из файла целого числа

```
fscanf( fi, "%d", &n );
```

Функция **feof(файловая-переменная)** возвращает 1 (истинное значение), если файл, открытый для считывания, закончился. Она возвращает 0, если из файла еще можно читать данные.

**Пример**

```
FILE * fi;
fi = fopen("test.txt","rt"); // rt означает открытие текстового файла на чтение
int n;
while( !feof(fi) ) {
fscanf( fi, "%d", &n );
ptintf("%d\n",n);
}
fclose(fi);
```

Для записи строки в файл используется функция **fputs(f1,**

**str).**

Чтение полной строки из текстового файла удобнее выполнить с помощью функции **fgets(str,n,f1).** Здесь параметр **n** означает максимальное количество считываемых символов, если раньше не встретится символ конца строки.

Обычно для обмена с двоичными файлами используются функции **fread и fwrite:**

**c_w = fwrite(buf, size_rec, n_rec, f1);**

Здесь

**buf** – указатель типа void* на начало буфера в оперативной памяти, из которого информация переписывается в файл;

**size_rec** – размер передаваемой порции в байтах;

**n_rec** – количество порций, которое должно быть записано в файл;

**f1** – указатель на блок управления файлом;

**c_w** – количество порций, которое фактически записалось в файл.

Считывание данных из двоичного файла осуществляется с помощью функции fread:

**c_r = fread(buf, size_rec, n_rec, f1);**

Здесь

**c_r** – количество порций, которое фактически прочиталось из файла;

**buf** – указатель типа void* на начало буфера в оперативной памяти, в который информация считывается из файла.

Двоичные файлы допускают не только последовательный обмен данными. Так как размеры порций данных и их количество, участвующее в очередном обмене, диктуются программистом, а не смыслом информации, хранящейся в файле, то имеется возможность пропустить часть данных или вернуться повторно к ранее обработанной информации. С другой стороны, функции fread/fwrite используются при работе с файлами прямого доступа, т.е. файлами, содержащими записи известного размера. В простейшем и наиболее используемом случае все записи имеют одинаковый размер. С помощью такого файла можно представить отдельную таблицу в реляционной базе данных. Название "файлы прямого доступа" отражает тот факт, что для доступа к конкретной i-й записи файла нет необходимости последовательно читать все предыдущие записи. Нужно переместить указатель текущей позиции на число байт, вычисляемое как произведение размера записи в байтах и номера записи, после чего можно сразу читать нужную запись (или записывать данные в эту запись).

Контроль за текущей позицией доступных данных в файле осуществляет система с помощью указателя, находящегося в блоке управления файлом. С помощью функции **fseek** программист имеет возможность переместить этот указатель:

**fseek(f1,delta,pos);**

Здесь

**f1** – указатель на блок управления файлом;

**delta** – величина смещения в байтах, на которую следует переместить указатель файла;

**pos** – позиция, от которой производится смещение указателя (0 или SEEK_SET – от начала файла, 1 или SEEK_CUR – от текущей позиции, 2 или SEEK_END – от конца файла)

## III. Файловые потоки в C++

Для работы с файлами необходимо подключить заголовочный файл <fstream>. В <fstream> определены несколько классов, в том числе ifstream – представляющий файловые потоки ввода и ofstream - файловые потоки вывода.

Файловый ввод/вывод аналогичен стандартному вводу/выводу, единственное отличие – это то, что ввод/вывод выполнятся не на экран, а в файл. Если ввод/вывод на стандартные устройства выполняется с помощью объектов cin и cout, то для организации файлового ввода/вывода достаточно создать собственные объекты, которые можно использовать аналогично потокам cin и cout.

**Запись в файл**

**Например,** необходимо создать текстовый файл и записать в него строку

Для этого необходимо проделать следующие шаги:

создать объект класса ofstream (если бы нужно было считывать из файла, то ifstream);

связать объект класса с файлом, в который будет производиться запись;

записать строку в файл;

закрыть файл.

```
#include <fstream>
using namespace std;
int main(int argc, char* argv[])
{
ofstream fout("cppstudio.txt"); // создаѐм объект класса of-
stream для записи и связываем его с файлом cppstudio.txt
```

```
fout << "Работа с файлами в С++"; // запись строки в файл
fout.close(); // закрываем файл
system("pause");
return 0;
}
```

**Чтение файла**

Для того чтобы прочитать файл понадобится выполнить те же шаги, что и при записи в файл с небольшими изменениями:

создать объект класса ifstream и связать его с файлом, в который будет производиться запись;

прочитать файл;

закрыть файл.

```
#include <fstream>
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
setlocale(LC_ALL, "rus"); // корректное отображение Кириллицы
char buff[50]; // буфер промежуточного хранения считываемого из файла текста
ifstream fin("cppstudio.txt"); // открыли файл для чтения
fin >> buff; // считали первое слово из файла
cout << buff << endl; // напечатали это слово
fin.getline(buff, 50); // считали строку из файла
fin.close(); // закрываем файл
cout << buff << endl; // напечатали эту строку
system("pause");
return 0;
}
```

Показано два способа чтения из файла, первый – используя операцию передачи в поток, второй – используя функцию getline(). В первом случае считывается только первое слово, а во втором случае считывается строка, длинной 50 символов. Но так как в файле осталось меньше 50 символов, то считываются символы включительно до последнего. Результат работы программы:

Для осуществления проверки, удалось ли открыть файл, используется функция **is_open()**, которая возвращает целые значения: 1 — если файл был успешно открыт, 0 — если файл от-

крыт не был.

**Например:**

```
if (!fin.is_open()) // если файл не открыт
cout << "Файл не может быть открыт!\n";
```

Также проверку можно осуществить и следующим образом.

```
ifstream input_file("FILENAME.DAT");
if (in.fail())
{
cout << "Ошибка открытия FILENAME.EXT" << endl;
exit(1);
}
```

Кроме того, для проверки того, находится ли поток в нормальном состоянии, можно использовать имя самого объекта, представляющего поток

```
if (!in)
{
cout << "Ошибка открытия FILENAME.EXT" << endl;
exit(1);
}
```

или

```
if (in)
{
// Все хорошо
} else
{
cout << "Ошибка открытия FILENAME.EXT" << endl;
exit(1);
}
```

Это возможно потому, что в классах потоков переопределены операции логического отрицания (!) и преобразования потока в нетипизированный указатель, которые явно (!) и неявно вызываются в предыдущих примерах.

**Режимы открытия файлов**

Режимы открытия файлов устанавливают характер использования файлов. Для установки режима в классе ios_base предусмотрены константы, которые определяют режим открытия файлов:

ios_base::in открыть файл для чтения
ios_base::out открыть файл для записи

ios_base::ate при открытии переместить указатель в конец файла

ios_base::app открыть файл для записи в конец файла

ios_base::trunc удалить содержимое файла, если он существует

ios_base::binary открытие файла в двоичном режиме

Режимы открытия файлов можно устанавливать непосредственно при создании объекта или при вызове функции open().

ofstream fout("cppstudio.txt", ios_base::app); // открываем файл для добавления информации к концу файла

fout.open("cppstudio.txt", ios_base::app); // открываем файл для добавления информации к концу файла

Режимы открытия файлов можно комбинировать с помощью поразрядной логической операции или |, например: ios_base::out | ios_base::trunc - открытие файла для записи, предварительно очистив его.

Объекты класса ofstream, при связке с файлами по умолчанию содержат режимы открытия файлов ios_base::out | ios_base::trunc. То есть файл будет создан, если не существует. Если же файл существует, то его содержимое будет удалено, а сам файл будет готов к записи. Объекты класса ifstream связываясь с файлом, имеют по умолчанию режим открытия файла ios_base::in - файл открыт только для чтения.

**Пример**

```
#include <fstream>
#include <iostream>
#include <cstdio>
#include <cstdlib>
using namespace std;
int main ()
{
ofstream out("test.txt");
for (int i=0; i<100; i++)
out<<i<<"\n";
out.close;
ifstream in("test.txt");
int a[100], i=0;
while (! in.eof())
{
cin>>a[i];
i++;
```

```
}
for (int j=0; j<i; j++)
cout <<a[j]<<"\n";
in.close;
return 0;
}
```

При работе с файловыми потоками можно использовать те же модификаторы потока, что и при работе с потоками стандартного ввода/вывода.

## 7.2. Задание к лабораторной работе (каждое задание реализовать как на С, так и на С++)

1. Напишите подпрограмму, подсчитывающую количество символов в указанном файле, открыв его вначале как текстовый, затем как двоичный. В текстовом режиме нужно посчитать также число строк файла.

2. Файл содержит как русские, английские буквы и другие символы. Написать программу, которая русские буквы из данного файла переписывает в отдельный новый файл, английские буквы в другой файл.

3. Объявить одномерный массив и заполнить его случайными числами. Записать в файл все положительные элементы массива и их сумму.

4. Файл f1 содержит последовательность целых положительных чисел в 10-й системе счисления. Вывести на экран содержимое файла f1. Записать в файл f2 четные числа из файла f1 в 8-ой системе счисления. Записать в файл f3 нечетные числа из файла f1 в 16-ой системе счисления. Вывести файлы f2 и f3 на экран (при этом выводится 10 символов, для продолжения вывода пользователь должен нажать букву ―n‖).

5. В процессе проведения некоторого спортивного турнира должна сохраняться информация о показателях каждой участвующей в турнире команды. Информация включает в себя название команды, количества выигранных, проигранных и сведенных вничью партий, а также общее количество набранных баллов. Напишите функции для сохранения в файле и считывания из файла таблицы текущего состояния турнира.

6. Создайте текстовое меню для управления информацией о спортивном соревновании. Должно поддерживаться следующее поведение: создание таблицы результатов, отображение таблицы, изменение таблицы, сохранение таблицы в файле, считывание таблицы из файла. Таблица должна содержать информацию, указанную в предыдущем задании.

## 7.3. Контрольные вопросы

1. Что такое файловый поток ввода/вывода?
2. Как создать файловый поток в Си?
3. В каком файле содержатся описания функций файлового ввода/вывода Си?
4. Какие режимы работы с файловыми потоками поддерживаются в Си?
5. Что вернет функция fopen при следующем обращении к ней:

```
FILE *f = fopen("abracadabra.abr","r");
```

если файл "abracadabra.abr" не существует?

6. Что вернет функция fopen при следующем обращении к ней:

```
FILE *f = fopen("abracadabra.abr","w");
```

если файл "abracadabra.abr" не существует?

7. Как задать режим работы файла, позволяющий и читать и записывать данные в файл?
8. Какой файл заголовков C++ содержит описание классов файловых потоков ввода? Вывода?
9. Как создать файловый поток ввода (вывода) в C++?
10. Как проверить состояние файлового потока в C++?
11. Как задать двоичный режим для файлового потока ввода в C++?
12. Что такое файл прямого доступа?
13. Как управлять указателем текущей позиции в файле (Си)?
14. Двоичный файл прямого доступа содержит вещественные числа двойной точности. Как быстро прочитать 120-е по счету число?
15. В чем разница между текстовыми и двоичными файлами в Си/C++?

### 7.4. Литература

1. Brian W. Kernighan, Dennis M. Ritchie. "The C programming language". Second edition – Prentice Hall. – 1988, 272 p.
2. Стэнли Б. Липпман. "Язык программирования C++. Вводный курс". – М.: Вильямс. – 2007, 896 с.
3. Г. Шилдт. "Самоучитель C++". – СПб.: БХВ-Петербург. – 2001, 688 с.
4. К.Арнольд, Дж.Гослинг, Д.Холмс. "Язык программирования Java". 3-е изд. – М.: Вильямс. – 2001, 624 с.

# 8. ЛАБОРАТОРНАЯ РАБОТА №8: КОНСТРУКТОРЫ И ДЕСТРУКТОР

### 8.1. Теория

#### I. Конструкторы класса.

Для инициализации (задания начального значения) объекта класса (как при статическом, так и при или динамическом создании объекта) для него вызывается специальный метод класса, называемый конструктором. Конструктор – это метод, имя которого в точности совпадает с именем класса. Для класса можно создать несколько конструкторов, тогда они будут отличаться списками своих аргументов. Если в описании класса ни один конструктор не задан явно, компилятор создаст конструктор без аргументов. Но если задать хотя бы один конструктор для своего класса, компилятор не станет создавать конструктор по умолчанию. Например,

```
class A {
    int a;
  public:
    A(int x) { a = x; } /* конструктор с одним аргументом типа int, конструктор по умолчанию не создается */
};

int main() {
x       A a1;         /* Ошибка. Нет конструктора без аргументов. */
    A a2(10); /* Правильно. Вызывается конструктор с аргументом типа int */
    return 0;
}
```

## II. Конструктор копирования.

В ситуациях, когда требуется создание копии объекта, т.е. когда новый объект создается на основе уже существующего объекта того же класса, вызывается конструктор специального вида – конструктор копирования. Это конструктор с одним аргументом – ссылкой на объект собственного класса. Например,

```
class A {
    int a;
  public:
    A() { a = 0; } /* конструктор без аргументов */
    A(A& x) {     /* конструктор копирования */
        a = x.a;
    }
};

int main() {
    A a1;       /* Вызывается конструктор без аргументов */
    A a2( a1 );   /* Вызывается конструктор копирования */
    A a3 = a1;   /* Вызывается конструктор копирования */
    return 0;
}
```

Если аргумент функции или метода – объект класса (но не ссылка на объект) то при вызове в ней (или в нем) создается новый локальный объект как копия переданного объекта, т.е. будет вызываться конструктор копирования. То же самое происходит, если функция или метод возвращают объект класса.

Если конструктор копирования для класса не задан явно, то компилятор создает его. Этот конструктор просто копирует значения полей старого объекта в соответствующие поля нового объекта. Это может приводить к проблеме повисших ссылок, если хотя бы одно поле класса указатель.

## III. Конструкторы преобразования.

Для того, чтобы разрешить преобразование объектов чужого класса или величин примитивных типов в объект нашего класса необходимо задать для него конструкторы преобразования. Конструктор преобразования – это конструктор с одним аргументом. Тип аргумента определяет тот тип, который может быть преобразован в объект класса. Например,

```
class A {
    int a;
```

```
   public:
      A(int x) { a = x; } /* конструктор с одним аргументом ти-
па int */
   };

   void f( A x ); /* прототип функции с одним аргументом –
объектом класса A */

   int main() {
      A a1(10); /* вызывается конструктор с одним аргументом
*/
      f( 5 );      /* вызывается конструктор преобразования */
      return 0;
   }
```

В этом примере при вызове функции f компилятор создаст временный объект класса A т.к. есть конструктор преобразования величины типа int в объект класса. Единственный конструктор в этом классе теперь выполняет две роли. Во-первых, это конструктор, вызываемый при создании новых объектов. Во-вторых, он задает преобразование целых значений в объекты класса. C++ позволяет объявлять конструкторы с одним аргументом так, чтобы их нельзя было использовать для преобразования типа. Для этого конструктор должен иметь модификатор explicit. Например,

```
   class A {
      int a;
   public:
      explicit A(int x) { a = x; }
   };

   void f( A x ); /* прототип функции с одним аргументом –
объектом класса A */

   int main() {
      A a1(10); /* вызывается конструктор с одним аргументом
*/
   x  f( 5 );      /* Ошибка. Не определено преобразование int
в A */
      return 0;
   }
```

**IV. Деструктор класса.**

Перед уничтожением объекта для него обязательно вызывается специальный метод класса, называемый деструктором. Имя деструктора начинается со значка тильда ~, после которого следует имя класса. Деструктор нельзя перегружать. Если он не задан для класса явно, то будет создан компилятором. Пример:

```
class Person {
    char *name;
    int age;
  public:
    Person( char *n, int a) { /* конструктор */
       name = new char[ strlen(n) + 1 ]; /* захват дополнительной памяти */
       strcpy( name, n );
       age = a;
    }
    ~Person() { /* деструктор */
       delete[] name; /* освобождение памяти, захваченной конструктором */
    }
};
```

## 8.2. Задание к лабораторной работе

1. Создать структуру "Студент" содержащую следующие поля:

- имя студента - произвольная длина
- отчество студента - произвольная длина
- фамилию студента - произвольная длина
- год рождения
- группа.

2. Определить конструктор для инициализации полей структуры со значениями по умолчанию. Определить конструктор копирования и деструктор. Написать тестовый пример.

3. Изменить в описании структуры ключевое слово struct на class. Запустить программу. Какие возникли проблемы? Почему? Как их исправить?

4. Написать интерфейсные функции доступа к полям класса (получить/задать значение поля).

5. Внести в конструкторы и деструктор выдачу сообщений на экран о том, какая функция была вызвана. Модифицировать функцию main следующим образом:

```
void main(void)
```

```
{
    cout<<"Вход в функцию main()"<<endl;
    ...
    <тело_main()>
    ...
    cout<<"Выход из функции main()"<<endl;
}
```

Выяснить время вызовов конструкторов и деструкторов.

6. Описать глобальную функцию Student test(Student s){return s;}. Вызвать ее в основной программе. Что произошло и почему?

7. Изменить передачу параметра ф-и test на передачу по ссылке. Что изменилось?

8. Изменить возврат результата ф-и test на передачу по ссылке. Что изменилось?

## 8.3. Контрольные вопросы

1. Каким образом инициализируются объекты класса в C++?
2. Может ли у класса быть несколько конструкторов?
3. Чем конструктор копирования отличается от других конструкторов класса?
4. Может ли конструктор копирования вызываться неявно?
5. Когда необходимо явно определять конструктор копирования для класса?
6. Как можно задать преобразование величины целого типа в объект нашего класса?
7. Как можно запретить использовать конструктор для преобразования типа?
8. Может ли у класса быть несколько деструкторов?
9. Когда вызывается деструктор класса?
10. Каково назначение деструктора?

## 8.4. Литература

1. Brian W. Kernighan, Dennis M. Ritchie. "The C programming language". Second edition – Prentice Hall. – 1988, 272 p.
2. Стэнли Б. Липпман. "Язык программирования С++. Вводный курс". – М.: Вильямс. – 2007, 896 с.
3. Г. Шилдт. "Самоучитель С++". – СПб.: БХВ-Петербург. – 2001, 688 с.
4. К.Арнольд, Дж.Гослинг, Д.Холмс. "Язык программирования Java". 3-е изд. – М.: Вильямс. – 2001, 624 с.

# 9. ЛАБОРАТОРНАЯ РАБОТА №9: ССЫЛКИ. КОПИРОВАНИЕ И ПРИСВАИВАНИЕ

## 9.1. Теория

### I. Создание и использование ссылок в C++.

Ссылка в C++ реализуется как скрытый константный указатель. Ссылка связывается с некоторым адресом и далее всегда остается связанной только с этим адресом. Однако этим адресом управляет не программист, а компилятор. При всяком обращении к ссылке компилятор обязательно проводит разадресацию и возвращает сам объект. Таким образом, мы всегда думаем о ссылке как об объекте, а не адресе. Например,

```
char ch = 'A';
char &rch = ch; /* ссылка на char связывается с переменной в момент создания */
rch = 'B'; /* результат такой же, как после ch = 'B'; */
```

### II. Назначение конструктора копирования.

В ситуациях, когда требуется создание копии объекта, т.е. когда новый объект создается на основе уже существующего объекта того же класса, вызывается конструктор специального вида – конструктор копирования. Это конструктор с одним аргументом – ссылкой на объект собственного класса. Например,

```
class A {
    int a;
  public:
    A() { a = 0; } /* конструктор без аргументов */
    A(A& x) {     /* конструктор копирования */
        a = x.a;
    }
};

int main() {
    A a1;       /* Вызывается конструктор без аргументов */
    A a2( a1 );   /* Вызывается конструктор копирования */
    A a3 = a1;   /* Вызывается конструктор копирования */
    return 0;
}
```

Если аргумент функции или метода – объект класса (но не ссылка на объект) то при вызове в ней (или в нем) создается новый локальный объект как копия переданного объекта, т.е. будет

вызываться конструктор копирования. То же самое происходит, если функция или метод возвращают объект класса.

### III. Конструктор копирования по умолчанию.

Если конструктор копирования для класса не задан явно, то компилятор создает его. Этот конструктор просто копирует значения полей старого объекта в соответствующие поля нового объекта. Это может приводить к проблеме внешней зависимости, в частности к проблеме повисших ссылок, если хотя бы одно поле класса указатель. В этом случае внешний адрес сохраняется в объекте класса и все изменения, выполненные вне объекта, скажутся на этом объекте. В таких случаях обычно нужно явно переопределить конструктор копирования для класса. Бывают ситуации, когда копирование объектов вообще недопустимо. Тогда нужно объявить конструктор копирования как закрытый, причем достаточно задать только объявление (прототип) а реализацию можно не задавать вообще.

```
class Socket {
    …
private:
    Socket(Socket&);
    …
};
```

### IV. Конструктор копирования и операция присваивания.

По умолчанию, присваивание также заключается в копировании значений полей одного объекта в соответствующие поля другого объекта этого же класса. Поэтому, как и в случае конструктора копирования, если хотя бы одно поле класса указатель, скорее всего, необходимо переопределить для него операцию присваивания.

```
class A {
    char *name;
public:
    A(char *n) { name = strdup(n); }
    ~A() { free(name); }
    A(A& obj) { name = strdup(obj.name); }
    A& operator=(const A& obj) {
        if( this == &obj ) return; // проверка на самоприсваивание
        free(name);
```

```
        name = strdup(obj.name);
    }
    ...
};
```

### 9.2. Задание к лабораторной работе

1. Для решения различных задач используются методы Монте-Карло, предполагающие применение массивов случайных чисел с большим количеством элементов. Размер массива становится известным во время выполнения программы, т.е. массив должен создаваться динамически. Создайте две функции для решения одной и той же задачи: динамическое создание и заполнение случайными числами массива указанного размера. Первая функция должна использовать возвращаемое значение для передачи пользователю сгенерированного массива, а вторая должна передавать массив через один из своих аргументов. Стандартная библиотека Си содержит функции int rand() и void srand(unsigned) для генерации псевдослучайных чисел (прототипы в файле stdlib.h).

2. Создать класс Person содержащий следующие поля:
   - фамилия и имя человека - произвольная длина
   - профессия человека - произвольная длина
   - возраст человека.

3. Определить конструктор для инициализации полей класса со значениями по умолчанию. Определить конструктор копирования, операцию присваивания и деструктор. Написать тестовый пример.

4. Написать интерфейсные функции доступа к полям класса (получить/задать значение поля).

5. Внести в конструкторы, операцию присваивания и деструктор выдачу сообщений на экран о том, какая функция была вызвана. Модифицировать функцию main следующим образом:

```
void main(void)
{
    cout<<"Вход в функцию main()"<<endl;
    ...
    <тело_main()>
    ...
    cout<<"Выход из функции main()"<<endl;
}
```

6. Задать глобальную функцию Person test(Person s) { return s; }

Вызвать ее в основной программе. Что произошло и почему?

7. Изменить передачу параметра функции test на передачу по ссылке.

Что изменилось?

8. Изменить возврат результата функции test на передачу по ссылке.

Что изменилось?

## 9.3. Контрольные вопросы

1. Чем ссылка в C++ отличается от указателя?
2. Чем конструктор копирования отличается от других конструкторов класса?
3. Может ли конструктор копирования вызываться неявно?
4. Когда необходимо явно определять конструктор копирования для класса?
5. Может ли аргумент конструктора копирования быть не ссылкой, а значением?
6. Как можно запретить копирование объектов класса?
7. Нужен ли для копирования объектов класса конструктор копирования, если для класса реализована операция присваивания?
8. Будет ли вызываться конструктор копирования класса A при вызове функции void f(A x)?
9. Будет ли вызываться конструктор копирования класса A при вызове функции void f(A& x)?
10. Будет ли вызываться конструктор копирования класса A при вызове функции A f()?
11. Будет ли вызываться конструктор копирования класса A при вызове функции A& f()?

## 9.4. Литература

1. Brian W. Kernighan, Dennis M. Ritchie. "The C programming language". Second edition – Prentice Hall. – 1988, 272 p.

2. Стэнли Б. Липпман. "Язык программирования C++. Вводный курс". – М.: Вильямс. – 2007, 896 с.

3. Г. Шилдт. "Самоучитель C++". – СПб.: БХВ-Петербург. – 2001, 688 с.

4. К.Арнольд, Дж.Гослинг, Д.Холмс. "Язык программирования Java". 3-е изд. – М.: Вильямс. – 2001, 624 с.

# 10. ЛАБОРАТОРНАЯ РАБОТА №10: ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ НА ПРИМЕРЕ ДВУСВЯЗНОГО СПИСКА

## 10.1. Теория

### I. Объектно-ориентированная реализация динамических структур данных.

В объектно-ориентированной парадигме каждое важное понятие решаемой задачи в общем случае должно быть представлено соответствующим классом. Например, если в программе нужно реализовать двунаправленный список, то этому понятию будет соответствовать класс с подходящим именем, скажем List. Но сам список в своей реализации использует такое важное понятие как узел списка, поэтому при реализации списка удобно создать еще один класс с именем, например, ListNode (узел списка). Этот класс нужен разработчику класса List, но нужен ли он программисту не создающему, а только использующему List? При использовании списка мы можем думать о тех данных, которые мы помещаем или извлекаем из списка, но совсем не интересуемся тем, как организовано хранение этих данные в классе List. Если это так, разработчик может спрятать от пользователя реализацию класса ListNode внутри реализации класса List, сделав последний вложенным закрытым (или защищенным) классом.

### II. Представление двунаправленного списка в программе на C++.

Разделяя описание и реализацию, при создании двунаправленного списка его описание помещают в файл заголовков, а реализацию в .cpp-файл, который компилируется и может предоставляться пользователю как часть статической или динамической библиотеки. Описание классов, реализующих двунаправленный список, может быть таким:

```
Файл list.h
class List {
    // закрытый вложенный класс
    class ListNode {
    public:
        int key; // уникальное для каждого узла значение
        char *data; // данные, хранящиеся в узле
```

```
        ListNode *prev, *next; /* указатели на предыдущий и
следующий узлы */
    };

    ListNode *first; // указатель на первый узел списка
public:
    List() : first(0) {}
    ~List() { del(); }
    void addData(int key, char *data); // добавить узел
    void removeData(int key); // удалить узел с указанным
ключом
    char *findData(int key); // вернуть данные по ключу
    void show(); // отобразить список в консольном окне
private:
    ListNode *findNode(int key); // поиск узла по ключу
    void del(); // удалить все узлы из списка
};
```

### III. Создание очереди и стека с помощью повторного использования кода.

Вместо того, чтобы создавать очередь или стек "с нуля", можно воспользоваться уже готовыми реализациями подходящих структур данных. Чаще всего очереди и стеки реализуют либо на основе массива, либо на основе списка. Организовать взаимодействие нового класса (стек, очередь) с уже имеющимся классом (например, List) можно по-разному. Это может быть отношение агрегирования, или наследования, отношение использования и т.д.

## 10.2. Задание к лабораторной работе

1. Создайте класс List, представляющий понятие "двунаправленный список".
2. Создайте класс, реализующий простое текстовое меню. Используйте его для тестирования созданного в предыдущем задании класса List.
3. Создайте классы стек и очередь на основе класса List с помощью агрегирования.
4. Создайте классы стек и очередь на основе класса List с помощью наследования.
5. Сравните реализации стека и очереди и сделайте вывод, какое отношение между классами агрегирование или наследование является правильным при построении очереди и стека на основе списка.

### 10.3. Контрольные вопросы

1. Какую дисциплину добавления/удаления элементов реализует очередь?
2. Какую дисциплину добавления/удаления элементов реализует стек?
3. Для чего можно использовать вложенный класс при реализации динамической структуры данных, такой как список?
4. Как организовать отношение между классами "агрегирование"?
5. Как организовать отношение между классами "наследование"?
6. Как запретить пользователю использовать стек или очередь как список, если эти классы реализованы на основе списка с помощью агрегирования?
7. Как запретить пользователю использовать стек или очередь как список, если эти классы реализованы на основе списка с помощью наследования?
8. Можно ли рассматривать построение стека или очереди на основе списка как "расширение"?
9. Как реализовать стек или очередь на основе вписка с помощью наследования на языках программирования не поддерживающих закрытое наследование (например, Java или C#)?
10. Какое отношение агрегирование или наследование следует предпочесть при реализации очереди или стека на основе списка?

### 10.4. Литература

1. Brian W. Kernighan, Dennis M. Ritchie. "The C programming language". Second edition – Prentice Hall. – 1988, 272 p.
2. Стэнли Б. Липпман. "Язык программирования С++. Вводный курс". – М.: Вильямс. – 2007, 896 с.
3. Г. Шилдт. "Самоучитель С++". – СПб.: БХВ-Петербург. – 2001, 688 с.

## 11. ЛАБОРАТОРНАЯ РАБОТА №11: ПЕРЕГРУЗКА ОПЕРАЦИЙ

### 11.1. Теория

#### I. Дружественные функции и дружественные классы.

Если функция объявлена в классе дружественной, то она получает такие же привилегии что и методы самого класса, т.е.

получает доступ не только к открытой, но и к защищенной и закрытой частям класса. Чтобы воспользоваться этими привилегиями функция должна каким-то образом получить объект класса: в качестве аргумента или создавая такой объект во время своей работы, т.к. скрытого параметра this у нее нет. Например,

```
class A {
    friend void fun(A x); // объявление внешней функции дру-
гом класса
    int a; // закрытое поле класса
  public:
    A(int x) { a = x; }
};
void fun(A x) { // реализация функции
cout<<"a="<<x.a // доступ к закрытому полю
      <<endl;
}
```

Дружественной функцией может быть объявлен метод другого класса, и даже другой класс в целом. В последнем случае все методы класса становятся дружественными функциями. Например,

```
class A;  // предварительное описание класса
class B {
  public:
void f(A& x);            // доступ к закрытому полю A
};
class C {};
class A {
friend void B::f(A&); // объявление дружественного метода
friend class C; // объявление дружественного класса
int a;
  public:
A(int x) { a = x; }
};
void B::f(A&) { cout<<x.a<<endl; }

int main() {
    A aObj(5);
    B bObj;
    bObj.f( aObj );
    return 0;
}
```

## II. Перегрузка операций в С++.

В С++ можно переопределить большинство значков операций заданных для стандартных типов для собственных классов. Например, если A это класс, а a1 и a2 объекты этого класса, то компилятор не знает, как выполнить сложение двух его объектов, т.е. значение выражения a1 + a2 может быть вычислено, только если для класса будет задана функция или метод, с алгоритмом такого вычисления. Такая функция (или такой метод) должна иметь строго определенное имя, состоящее из зарезервированного в С++ слова operator и значка операции (через пробел или без пробела, за исключением операций new и delete, для которых пробел обязателен). В общем случае выражение a1 + a2 компилятор трактует либо как вызов метода для объекта a1

a1.operator+( a2 ),

либо как вызов внешней функции с двумя аргументами a1 и a2

operator+(a1,a2).

Общие правила перегрузки операций в С++ таковы:

1. Нельзя задать новый значок операции.
2. Нельзя изменить арность (число аргументов), приоритет и ассоциативность операции.
3. Нельзя переопределить операцию для стандартного типа.
4. Нельзя переопределять операции "запятая", sizeof, ::, ., .*, ?:, typeid, throw, dynamic_cast<>, static_cast<>, const_cast<>, reinterpret_cast<>.

Некоторые другие ограничения рассмотрены ниже.

## III. Перегрузка операций с помощью методов класса.

Операции =, [], (), -> могут быть перегружены только методами класса. Если перегруженная операция реализована как метод, то ее единственный (для унарных операций) или первый/левый аргумент (для бинарных операций) доступен методу через скрытый аргумент this. Например,

```
class A {
    int a;
  public:
    A(int x) { a = x; }
    int getA() { return a; }
    void setA(int x) { a = x; }
    A operator+(A& right) { /* перегруженная операция для
сложения объектов класса A */
        return A( a + right.a );
```

```
    }
};

int main() {
    A a1(1), a2(2), a3(3);
    a1.setA( 5 );
a2.setA( 10 );
a3 = a1 + a2;  // a3 = a1.operator+(a2);
    return 0;
}
```

### IV. Перегрузка операций с помощью внешних функций.

Бывают такие виды операций, когда их перегрузка с помощью методов невозможна. Например, класс А рассмотренный выше позволяет попарно складывать свои объекты, но не позволяет добавить к целому числу объект класса. Вообще, если левый аргумент операции не является объектом нашего класса, такую операцию нельзя перегрузить с помощью метода нашего класса. Приходится использовать внешнюю функцию и, что дать ей прямой доступ к закрытым полям класса, такую функцию часто объявляют дружественной классу. Характерным примером, кроме приведенного выше, является также желание переопределить для класса операцию помещения в поток (или извлечения из потока). Левым аргументом такой операции должен быть поток, т.е. объект чужого класса ostream. Например,

```
#include <iostream>

using namespace std;

class A {
    friend ostream& operator<<(ostream&,A&); /* переопре-
деленная операция помещения в поток */
    int a;
  public:
    A(int x) { a = x; }
};

ostream& operator<<(ostream& o, A& x) {
    return o<<x.a;
}
```

```
int main() {
    A aObj( 10 );
    cout<<aObj<<endl;  // помещение объекта в поток
    return 0;
}
```

## 11.2. Задание к лабораторной работе

1. Описать класс Test с защищенными числовым полем W и функцией Z, которая выводит сообщение «Это закрытая функция класса Test". Написать конструктор для инициализации объектов класса Test с одним параметром, принимающим по умолчанию значение 1. Объявить другом класса функцию fun, которая не возвращает значений и получает указатель на объект типа Test.
2. Описать на внешнем уровне функцию fun, которая выводит на экран значение параметра W и вызывает из класса Test функцию Z.
3. В функции main описать переменную класса Test (без явной инициализации) и применить к ней функцию fun.
4. Создать класс с перегруженной операцией присваивания.
5. Придумать и реализовать программу - пример использования двух классов А и В, в которой А друг В.

## 11.3. Контрольные вопросы

1. Что такое дружественная функция?
2. Можно ли объявить некоторую функцию дружественной вне описания класса? Почему?
3. Что такое дружественный класс?
4. В чем отличие операции присваивания от конструктора копирования?
5. Назначение друзей класса?
6. Какие операции можно реализовать только методами класса?
7. В каких случаях операцию можно перегрузить только с помощью внешней функции?
8. Можно ли операцию вставки в поток реализовать как функцию-член класса?
9. Как переопределить постфиксную и префиксную формы операции инкремента?
10. Для каких классов удобно переопределить операцию индексирования?

### 11.4. Литература

1. Brian W. Kernighan, Dennis M. Ritchie. "The C programming language". Second edition – Prentice Hall. – 1988, 272 p.
2. Стэнли Б. Липпман. "Язык программирования C++. Вводный курс". – М.: Вильямс. – 2007, 896 с.
3. Г. Шилдт. "Самоучитель C++". – СПб.: БХВ-Петербург. – 2001, 688 с.
4. К.Арнольд, Дж.Гослинг, Д.Холмс. "Язык программирования Java". 3-е изд. – М.: Вильямс. – 2001, 624 с.

# 12. ЛАБОРАТОРНАЯ РАБОТА №12: СТАТИЧЕСКИЕ ПОЛЯ И МЕТОДЫ КЛАССА

### 12.1. Теория

**I. Статические поля класса в C++.**

Если в классе созданы поля, то каждый объект класса будет обладать собственными копиями всех полей, поэтому такие поля называют полями экземпляра класса. Можно объявлять такие поля, которые будут принадлежать классу в целом. Такие поля будут существовать в единственном экземпляре независимо от того, сколько объектов класса создано. Они будет существовать, даже если нет ни одного объекта класса. Чтобы объявить поле класса надо задать при его описании модификатор static. Такое поле будет храниться в сегменте данных программы, вместе с глобальными и статическими переменными. Кроме того C++ требует, чтобы эти поля явно были созданы вне описания класса. По умолчанию статические поля инициализируются нулевыми значениями. Например,

```
class A {
   public:
int a1;          /* описание поля экземпляра класса */
static int a2;      /* описание поля класса в целом */
static int a3;      /* описание поля класса в целом */
};

int A::a2; /* объявление статического поля, инициализация нулем по умолчанию */
int A::a3 = 100; /* объявление статического поля, явная инициализация */

int main() {
    A::a2 = -1; /* статиче-  ское  поле  существует,  когда
```

```
нет ни одного объекта класса */
        A x; /* объект класса */
        x.a2 = -10; /* обращение к статическому полю через
объект */
        return 0;
    }
```

## II. Статические методы класса в С++.

Методы, объявленные в классе, могут быть вызваны только для какого-нибудь объекта этого класса ("привязаны" к объекту), поэтому их называют методами экземпляра класса. Адрес объекта, для которого вызван метод, передается последнему через скрытый параметр и доступен в теле метода как указатель с именем this. Если же при объявлении метода задан модификатор static, то такой метод не будет привязан к объекту класса, а станет методом класса в целом. Так как при вызове статического метода может не задаваться объект, у него не может быть скрытого аргумента – адреса объекта и, поэтому, в теле такого метода не определено имя this. С другой стороны, статический метод можно вызывать даже тогда, когда нет ни одного объекта класса. Например,

```
    class A {
        int a1;
        static int a2;
      public:
        A(int x) { a1 = x; }
        static void stFun() {
    x       a1 = 1; /* Ошибка. Нет this, поэтому нет доступа к не-
статическим полям */
            a2 = 1; /* Есть доступ */
    }
    };
    int A::a2;

    int main() {
        A::stFun(); /* можно вызвать, хотя нет ни одного объекта
A */
        A x(10);
        x.stFun();   /* так тоже можно */
        return 0;
    }
```

## III. Статические поля и методы в Java.

Статические поля и методы в Java имеют тот же смысл, что

и в C++. Но в Java статические поля не только описываются, но и определяются внутри описания класса. Например,

```
public class A {
    public static int a = 20;
}
```

Кроме того, для инициализации статических полей в Java можно использовать статические блоки инициализации. Блоки инициализации – это блоки кода, которые могут содержать также локальные описания. Например,

```
public class A {
    public static int[] a;
    static {  /* статический блок инициализации */
        a = new int[20];
        for(int i=0; i<20; ++i) a[i] = i + 1;
    }
}
```

Блоков инициализации может быть несколько. Компилятор объединяет их в один блок в том порядке, в котором они заданы в описании класса. Этот блок выполняется при загрузке класса.

В Java нет внешних функций, поэтому точкой входа в программу должен быть статический метод (main), чтобы его можно было вызвать тогда, когда еще нет ни одного объекта.

```
public class A {
    public static void main(String[] args) {
        System.out.println("Hello!");
    }
}
```

## 12.2. Задание к лабораторной работе

1. Создайте класс, который содержит счетчик созданных объектов. Напишите программу-тест, в которой проверяется, сколько объектов класса создано при входе в функцию main, после статического создания массива объектов, после динамического создания объекта, после удаления динамического объекта.

2. Создайте класс с закрытыми конструкторами и деструктором. Реализуйте методы для создания и уничтожения объектов класса. Напишите программу-тест.

3. Создайте класс, для которого возможно создание только одного объекта.

4. Создайте класс на Java со статическими полями и методами. Инициализируйте статические поля в статическом блоке инициализации.

### 12.3. Контрольные вопросы

1. В чем отличие статических и нестатических полей?
2. Где размещаются статические поля класса? Нестатические поля?
3. Как можно проинициализировать статическое поле в C++?
4. Как можно проинициализировать статическое поле в Java?
5. Когда выполняются статические блоки инициализации?
6. Для решения каких задач можно использовать статические поля?
7. Чем статический метод отличается от нестатического?
8. Можно ли в теле статического метода получить доступ к нестатическим полям класса?
9. Можно ли в теле статического метода вызвать нестатический метод класса?
10. Может ли точка входа в Java-приложение быть нестатическим методом? Почему?

### 12.4. Литература

1. Brian W. Kernighan, Dennis M. Ritchie. "The C programming language". Second edition – Prentice Hall. – 1988, 272 p.
2. Стэнли Б. Липпман. "Язык программирования C++. Вводный курс". – М.: Вильямс. – 2007, 896 с.
3. Г. Шилдт. "Самоучитель C++". – СПб.: БХВ-Петербург. – 2001, 688 с.
4. К.Арнольд, Дж.Гослинг, Д.Холмс. "Язык программирования Java". 3-е изд. – М.: Вильямс. – 2001, 624 с.

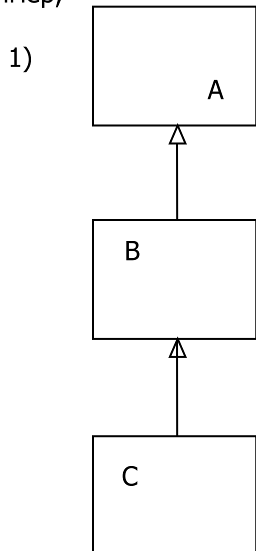
# 13. ЛАБОРАТОРНАЯ РАБОТА №13: НАСЛЕДОВАНИЕ В С++

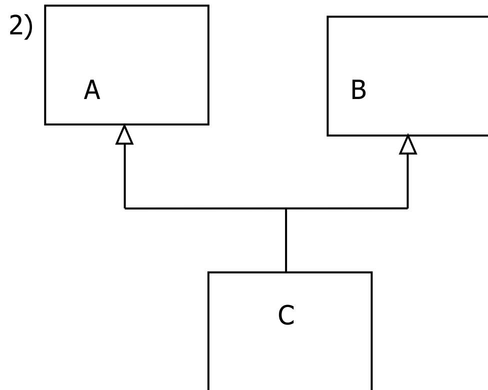
### 13.1. Теория

**I. Отношение расширения (наследования).**

Между классами можно установить отношение, в котором некоторые классы будут выступать в качестве базовых для построения других, производных от них классов. При этом производные классы получают все состояния и все поведение базовых классов и могут добавить свои состояния и свое поведение. Поэтому отношение и называется отношением расширения. Связь между классами становится родственной, поэтому базовые классы

называют также родительскими или классами предками, а производные наследниками, дочерними ил классами потомками. На UML-диаграмме классов отношение наследования задается стрелками, направленными от производных к базовым классам. Например,




На обеих диаграммах у класса C есть два базовых класса A и B. Но на схеме 1) класс B является прямым базовым классом для C, а класс A непрямым. На схеме 2) оба класса, и B и A прямые базовые классы для C. поэтому схема 1) задает так называемое единичное, а схема 2) множественное наследование.

## II. Виды наследования в C++.

В C++ общий синтаксис объявления производного класса при единичном наследовании таков:

class *Имя_производного_класса* : [*вид_наследования*] *Имя_базового_класса* {
описания полей и методов производного класса
};

Если вид наследования не задан явно, то, в случае, если класс создавался с помощью ключевого слова class, по умолчанию задается закрытое наследование, а если с помощью ключевого слова struct, то открытое. Можно задать открытое (public), закрытое (private) или защищенное (protected) наследование. Вид наследования не влияет на доступ методов производного класса к полям и методам унаследованным от базового класса. Он опреде-

ляет доступ к этим полям и методам из других классов или внешних функций. При открытом наследовании доступ определяется описанием базового класса. При закрытом наследовании все описания базового класса становятся закрытыми для тех классов и функций, которые используют объекты производного класса. При защищенном наследовании закрытая часть базового класса остается закрытой, а остальные становятся защищенными. Например,

```
class Base {
  private:    int privB;
  protected: int protB;
  public: int pubB;
};

class Derived : protected Base {
  public:
   void test() {
   x  privB = 1;   // Ошибка. Нет доступа к закрытому полю
      protB = 2;   // Есть доступ, как у наследника
      pubB = 3;   // Есть доступ, как у всех
   }
};

class Derived2 : protected Derived {
  public:
   void test() {
   x  privB = 1;   // Ошибка. Нет доступа к закрытому полю
      protB = 2;   // Есть доступ, как у наследника
      pubB = 3;   // Есть доступ, как у всех
   }
};

int main() {
   Derived d;
x  d.privB = 1;   // Ошибка. Нет доступа к закрытому полю
x  d.protB = 2;   // Ошибка. Нет доступа т.к. не наследник
x  d.pubB = 3;   // Ошибка. Нет доступа т.к. не наследник
   return 0;
}
```

### III. Множественное наследование.

При множественном наследовании у класса наследника есть не менее двух прямых предков. При объявлении наследования

вид наследования задается индивидуально для каждого базового класса. Например,

```
class Base1 {};
class Base2 {};
class Derived : public Base1, protected Base2 {};
```

При создании объекта производного класса, прежде всего вызываются конструкторы базовых классов в том порядке, в котором они заданы в описании класса наследника, потом конструктор производного класса. При уничтожениии объекта производного класса деструкторы вызываются в обратном порядке. Чтобы организовать ромбовидное наследование необходимо использовать виртуальные базовые классы:

```
class A {};
class B : virtual public A {};
class D : virtual public A {};
class C : public B, public C {};
```

## 13.2. Задание к лабораторной работе

1. Создать базовый класс Base, в котором описать в разделе public поле i типа int, в разделе protected поле l типа long, в разделе private поле d типа double. Написать конструктор, инициализирующий поля i, l и d тремя задаваемыми значениями.

2. Создать класс Derived, производный от класса Base (наследование типа public), в котором в разделе private описано поле f типа float. В классе Derived создать конструктор без параметров и конструктор с четырьмя параметрами для инициализации всех полей объекта.

3. В функции main описать неинициализированный объект класса Derived и откомпилировать программу. Если есть проблемы, то устранить их. Вывести размеры типов Base и Derived и объяснить результаты.

4. Описать инициализированный объект класса Derived. Продемонстрировать, инициализацию каких полей, унаследованных от класса Base, можно выполнять с помощью присваивания непосредственно в конструкторе класса Derived. Для исследования можно вносить необходимые изменения в конструкторы классов Base и Derived.

5. Перегрузить операцию вставки в поток для объектов класса Derived таким образом, чтобы выводились адреса и значения всех полей объекта. К каким полям, унаследованным от класса Base, нет доступа? Для снятия проблемы добавить в классе Base необходимые интер- фейсные функции. Создав объ-

ект класса Derived, исследовать размещение полей в памяти. В отчете привести схематическую структуру объекта.

6. Описать класс Derived1, производный (public) от класса Derived и не имеющий новых полей. В классе описать конструктор со всеми необходимыми параметрами (сколько их нужно?). Класс имеет общедоступную функцию void foo(), которая модифицирует значения полей, унаследованных от базового класса (i++; l+=1;). Откомпилировать программу. Заменить тип наследования Derived от Base на private и вновь откомпилировать программу. Какая возникла проблема? Для ее решения использовать возможность восстановления уровня доступа к компонентам базового класса.

7. Вернуть для Derived тип наследования public. На глобальном уровне и в классах Base и Derived описать функции void ff(), которые сообщают о своей принадлежности к классу или глобальному уровню. В функции foo класса Derived1 добавить вызовы всех трех функций ff. В каких разделах классов Base и Derived нужно описать функции ff, чтобы они были доступны в Derived1? Проверить работу программы, вызвав функцию foo для какого-либо объекта класса Derived1.

8. Оставить в функции Derived::foo только один вызов в виде ff(); и проверить работу программы в следующих вариантах. Вначале функция ff определена в классах Derived, Base и на глобальном уровне. Затем ее описание убираем вначале из класса Derived, а затем из классов Derived и Base. Как в каждом случае это отражается на работе программы?

9. Класс Base1, имеет одно закрытое поле i целого типа. Первый конструктор не имеет параметров и обнуляет i. Второй имеет один параметр типа int, используемый для инициализации i произвольными значениями. Класс имеет две общедоступные интерфейсные функции void put(int) и int get(void), которые позволяют изменить или прочесть значение i. Класс Base2, имеет одно закрытое поле - массив name из 20 элементов. Первый конструктор не имеет параметров и инициализирует поле name словом "Пусто". Второй имеет один параметр типа char*, используемый для инициализации name значениями символьных строк. Класс имеет две общедоступные интерфейсные функции void put(char*) и char* get(void), которые позволяют изменить или прочесть значение name. Класс Derived имеет одно закрытое поле ch типа char. Первый конструктор не имеет параметров и присваивает ch значение 'V' (от void - пустой). Второй конструктор имеет три параметра типов char, char* и int, используемые для инициализации соответственно полей ch, name и i. Класс имеет две общедоступ-

ные интерфейсные функции void put(char) и char get(void), которые позволяют изменить или прочесть значение ch. Кроме того, в нем объявляется как дружественная операция вставки в поток вывода, которая выводит на экран значения i, name и ch.

а). В функции main описать переменную типа Derived без инициализации и вывести ее значение с помощью перегруженной операции вставки в поток. Выяснить порядок вызова конструкторов.

б). Описать другую переменную класса Derived, инициализировав ее явно некоторыми значениями. Вывести значение этой переменной на экран и проанализировать порядок вызова конструкторов.

в). В конструкторе класса Derived с параметрами изменить порядок вызова конструкторов базовых классов. Проверить, как это отразилось на работе программы и почему.

г). Изменить порядок наследования базовых классов в описании класса Derived и проверить, как это отразилось на работе программы.

## 13.3. Контрольные вопросы

1. К каким полям, унаследованным от базового класса, нет доступа в производном классе?
2. Какие поля производного класса обязательно нужно инициализировать с помощью конструктора базового класса?
3. На что влияет вид наследования?
4. Как восстановить уровень доступа к компонентам базового класса для пользователей производного класса, если наследование закрытое?
5. Почему в языках программирования Java и C# наследование может быть только открытым?
6. Как отличить единичное наследование от множественного?
7. Могут ли при множественном наследовании отличаться виды наследования для разных базовых классов?
8. Как организовать ромбовидное наследование?

## 13.4. Литература

1. Brian W. Kernighan, Dennis M. Ritchie. "The C programming language". Second edition – Prentice Hall. – 1988, 272 p.
2. Стэнли Б. Липпман. "Язык программирования C++. Вводный курс". – М.: Вильямс. – 2007, 896 с.
3. Г. Шилдт. "Самоучитель C++". – СПб.: БХВ-Петербург. –

2001, 688 с.
4. К.Арнольд, Дж.Гослинг, Д.Холмс. "Язык программирования Java". 3-е изд. – М.: Вильямс. – 2001, 624 с.

# 14. ЛАБОРАТОРНАЯ РАБОТА №14: ОБРАБОТКА ИСКЛЮЧЕНИЙ

## 14.1. Теория

### I. Исключительные ситуации.

Во время выполнения программы могут возникать различные нестандартные ситуации. Это могут быть ошибки, вызванные внешними или внутренними причинами, или ситуации, когда в некоторой точке программы нет достаточной информации для выбора правильного решения, но в другой ее точке эта информация есть. Такие ошибочные и неопределенные состояния называют исключительными. Стандартная схема обработки исключений предполагает, что программа, столкнувшись с исключительной ситуацией, генерирует исключение, при этом происходит передача управления в ту точку, где есть обработчик такой ситуации. Исключения различаются по своему типу. В C++ любой допустимый тип можно использовать как тип исключения. Соответственно и обработчиков прерываний, в общем случае, в программе несколько – каждый для своего типа исключения. Однако некоторые типы являются более "широкими" чем другие. Например, если тип одного исключения представлен базовым классом, а другого производным от базового, то обработчик исключения базового типа может обрабатывать и исключения производного типа. Кроме того, нетипизированный указатель является более широким, чем любой типизированный. Наконец, можно создать обработчик, способный обрабатывать исключения любого типа. Компилятор выбирает первый подходящий обработчик исключений, поэтому, если обработчиков исключения в одной точке программы указано несколько они должны задаваться в порядке от более специализированных (производных) к более широким (базовым).

### II. Обработка исключений.

Код, в котором могут возникать исключительные ситуации, нужно размещать в блоках try, тогда будет сохраняться необходимая информация для правильной раскрутки стека при обработке исключений. Собственно же обработка производится в подходящем блоке catch. Например,

```
class DivideException {
    int a; // значение числителя
public:
    DivideException(int x) : a(x) {}
    int getA() { return a; }
};
int divide(int x, int y); // может выбрасывать исключение DivideException
int main() {
    int a, b, c;
    while (1 ) {
cout << "Задайте значения a, b: ";
        cin >> a >> b;
        try {
            c = f(a,b); /* вызываем в блоке try, так как может генерировать исключение */
            cout << a << '/' << b << '=' << c << endl;
            break;
} catch(DivideException) {
    cout << "Попробуйте еще раз." << endl;
}
}
return 0;
}
```

Хотя для реализации обработчика исключения достаточно задать только тип исключения, из точки генерации исключения можно передавать и дополнительную информацию, характеризующую это исключение. Например, класс DivideException сохраняет информацию о значении числителя в ситуации, когда знаменатель был равен нулю. Эту информацию можно использовать в обработчике исключения, например

```
try {
    c = f(a,b);
    cout << a << '/' << b << '=' << c << endl;
    break;
} catch(DivideException de) {
cout << "Не могу разделить " << de.getA() << " на ноль." << endl;
cout << "Попробуйте еще раз." << endl;
}
```

**III. Генерация (выбрасывание) исключения.**

Для генерации исключения используется операция throw

выражение. Тип выражения определяет тип исключения. Например,

```
int divide(int x, int y) {
if( !y ) throw DivideException(x);
return x/y;
}
```

Если исключение не удается обработать в одном блоке catch, в нем можно повторно сгенерировать это же исключение, тогда окончательная обработка должна проводиться в блоке catch того же типа внешнего блока try. Например,

```
void f1(int x) {
try {
    if( x < 0 ) throw 0; // исключение типа int
…
}
void f2(int x) {
try {
    f1(x);
    …
} catch( int ) {
    … // частичная обработка исключения
    throw; // повторная генерация того же исключения
    …
}
}
int main() {
int a;
cin >> a;
try {
    f2(a);
    …
} catch( int ) {
    … // окончательная обработка исключения
}
}
```

### 14.2. Задание к лабораторной работе

1. Создать класс Array, представляющий понятие массива фиксированного размера. Переопределить операцию индексирования так, чтобы при выходе индекса за допустимые границы генерировалось исключение.

2. Создать класс целых чисел IntegerRange с ограниченным диапазоном (по умолчанию от 0 до 100). Предусмотреть воз-

можность задания интервала в конструкторе. Перегрузить операцию присваивания и арифметические операции. При выполнении любой из этих операций возможен выход за границы интервала, следовательно, при выходе должно выбрасываться исключение. Исключение должно быть представлено классом OutOfRangeException.

3. Создать класс положительных целых чисел PositiveInteger как класс наследник IntegerRange. Создать классы исключений LeftBoundException и RightBoundException, представляющие, соответственно, события выхода значения за левую и правую границы. Организуйте каскадный вызов исключений.

4. Используйте при создании класса PositiveInteger не наследование, а агрегирование.

## 14.3. Контрольные вопросы

1. Что такое исключительная ситуация?
2. Как различаются исключения в программе на С++?
3. Для чего нужны блоки try?
4. Важен ли порядок следования блоков catch?
5. Как передать в блок catch информацию из точки генерации исключения?
6. Как генерируются исключения в С++?
7. Как выглядит блок catch, способный обрабатывать исключения любого типа?

## 14.4. Литература

1. Brian W. Kernighan, Dennis M. Ritchie. "The C programming language". Second edition – Prentice Hall. – 1988, 272 p.

2. Стэнли Б. Липпман. "Язык программирования С++. Вводный курс". – М.: Вильямс. – 2007, 896 с.

3. Г. Шилдт. "Самоучитель С++". – СПб.: БХВ-Петербург. – 2001, 688 с.

4. К.Арнольд, Дж.Гослинг, Д.Холмс. "Язык программирования Java". 3-е изд. – М.: Вильямс. – 2001, 624 с.