Лабораторная работа №4. Работа с памятью. Работа с файлами.

Необходимо написать многофайловую программу, в которой применяются алгоритмы обработки некорректного пользовательского ввода, используется динамическое выделение памяти и происходит считывание/вывод данных из/в файл(а).

Теоретическая часть

Указатели

Указатель — это переменная, значением которой является адрес ячейки памяти.

```
int *iPtr; // указатель на значение типа int
double *dPtr; // указатель на значение типа double

int* iPtr3; // корректный синтаксис (допустимый, но не желательный)
int * iPtr4; // корректный синтаксис (не делайте так)

int *iPtr5, *iPtr6; // объявляем два указателя для переменных типа int
```

Оператор взятия адреса &

При выполнении инициализации переменной, ей автоматически присваивается свободный адрес памяти, и, любое значение, которое мы присваиваем переменной, сохраняется по этому адресу в памяти. Например:

```
#include <iostream>
int main()
{
    int a = 7;
    std::cout << a << '\n'; // выводим значение переменной a
    std::cout << &a << '\n'; // выводим адрес памяти переменной a
    return 0;
}
```

Возможный результат :

```
7
0x7fff02f1f3a4
```

*Оператор разыменования **

Оператор разыменования `*` позволяет получить значение по указанному адресу:

```
#include <iostream>
int main()
{
    int a = 7;
    std::cout << a << '\n'; // выводим значение переменной a
    std::cout << &a << '\n'; // выводим адрес переменной a
    std::cout << *&a << '\n'; /// выводим значение ячейки памяти переменной a
    return 0;
}
```

Возможный результат:

```
7
0x7ffe0152bd04
7
```

Присваивание значений указателю

```
int value = 5;
int *ptr = &value; // инициализируем ptr адресом значения переменной
```

Приведённое выше можно проиллюстрировать следующим образом, рисунок 4.1:

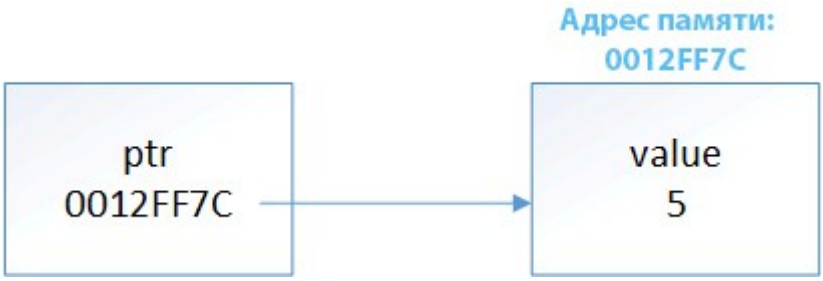

Рис. 4.1 Указатель, адрес памяти, значение

Тип указателя должен соответствовать типу переменной, на которую он указывает:

```
int iValue = 7;
double dValue = 9.0;

int *iPtr = &iValue; // ок
double *dPtr = &dValue; // ок
iPtr = &dValue; /* неправильно: указатель типа int не может указывать на адрес
переменной типа double */
dPtr = &iValue; /* неправильно: указатель типа double не может указывать на
адрес переменной типа int */
```

Следующее не является допустимым:

```
int *ptr = 7;
```

Указатели могут содержать только адреса, а целочисленный литерал `7` не имеет адреса памяти. Язык C++ также не позволит вам напрямую присваивать адреса памяти указателю:

```
double *dPtr = 0x0012FF7C; /* не ок: рассматривается как присваивание
целочисленного литерала */
```

Оператор адреса возвращает указатель

Оператор адреса & не возвращает адрес своего операнда в качестве литерала. Вместо этого он возвращает указатель, содержащий адрес операнда, тип которого получен из аргумента:

```
#include <iostream>
#include <typeinfo>
int main()
{
	int x(4);
	std::cout << typeid(&x).name();
```

```
	return 0;
}
```

Результат выполнения программы (указатель типа int):

Pi

Разыменование указателей

Как только у нас есть указатель, указывающий на что-либо, мы можем его разыменовать, чтобы получить значение, на которое он указывает. Разыменованный указатель — это содержимое ячейки памяти, на которую он указывает:

```
#include <iostream>

int main()
{
	int value = 5;
	std::cout << &value << std::endl; // выводим адрес value
	std::cout << value << std::endl; // выводим содержимое value
	int *ptr = &value; // ptr указывает на value
	std::cout << ptr << std::endl; /* выводим адрес, который хранится в ptr, т.е. &value */
	std::cout << *ptr << std::endl; /* разыменовываем ptr (получаем значение на которое указывает ptr) */

	return 0;
}
```

Результат:

```
0x7ffd986751dc
5
0x7ffd986751dc
5
```

Указатели должны иметь тип данных. Без типа указатель не знал бы, как интерпретировать содержимое, на которое он указывает (при разыменовании). Также, поэтому и должны совпадать тип указателя с типом переменной.

Нулевые указатели

Нулевое значение (или «значение `null`») — это специальное значение, которое означает, что указатель ни на что не указывает. Указатель, содержащий значение `null`, называется нулевым указателем.

```
int *ptr(0);  // ptr теперь нулевой указатель

int *ptr1; // ptr1 не инициализирован
ptr1 = 0; // ptr1 теперь нулевой указатель

#include <iostream>

int main()
{
	double *ptr(0);
```

```
	if (ptr)
		std::cout << "ptr is pointing to a double value.";
	else
		std::cout << "ptr is a null pointer.";

	return 0;
}
```

Ключевое слово nullptr

Использование `0` в качестве аргумента-литерала может привести к проблемам, так как компилятор не сможет определить, используется ли нулевой указатель или целое число `0`:

```
doAnything(0); /* является ли 0 аргументом-значением или аргументом-указателем? (компилятор определит его как целочисленное значение) */
```

Для решения этой проблемы в C++11 ввели новое ключевое слово `nullptr`, которое также является константой `r-value`.

```
int *ptr = nullptr; /* примечание: ptr по-прежнему остаётся указателем типа int, просто со значением null (0) */
```

Язык C++ неявно преобразует `nullptr` в соответствующий тип указателя. Таким образом, в вышеприведённом примере, `nullptr` неявно преобразуется в указатель типа `int`, а затем значение `nullptr` присваивается `ptr`.
`nullptr` также может использоваться для вызова функции (в качестве аргумента-литерала):

```
#include <iostream>

void doAnything(int *ptr)
{
    if (ptr)
        std::cout << "You passed in " << *ptr << '\n';
    else
        std::cout << "You passed in a null pointer\n";
}

int main()
{
    doAnything(nullptr); /* теперь аргумент является точно нулевым указателем, а не целочисленным значением */

    return 0;
}
```

Тип данных std::nullptr_t в C++11

В C++11 добавили новый тип данных `std::nullptr_t`, который находится в заголовочном файле `cstddef`. `std::nullptr_t` может иметь только одно значение — `nullptr`! Это полезно в одном случае. Если вам нужно написать функцию, которая принимает аргумент `nullptr`, то какой тип параметра нужно использовать? Правильно! `std::nullptr_t`. Например:

```
#include <iostream>
#include <cstddef> // для std::nullptr_t

void doAnything(std::nullptr_t ptr)
{
```

```
    std::cout << "in doAnything()\n";
}

int main()
{
    doAnything(nullptr); // вызов функции doAnything() с аргументом типа
std::nullptr_t
    return 0;
}
```

Передача по адресу

Тот факт, что массивы распадаются на указатели при передаче в функции, объясняет основную причину, по которой изменение массива в функции приведёт к изменению фактического массива.

```
#include <iostream>
// Параметр ptr содержит копию адреса массива
void changeArray(int *ptr)
{
    *ptr = 5; /* поэтому изменение элемента массива приведет к изменению
фактического массива */
}
int main()
{
    int array[] = { 1, 2, 3, 4, 4, 9, 15, 25 };
    std::cout << "Element 0 has value: " << array[0] << '\n';
    changeArray(array);
    std::cout << "Element 0 has value: " << array[0] << '\n';
     return 0;
}
```

Результат выполнения программы:

```
Element 0 has value: 1
Element 0 has value: 5
```

- При вызове функции `changeArray()`, массив распадается на указатель, а значение этого указателя (адрес памяти первого элемента массива) копируется в параметр `ptr` функции `changeArray()`. Хотя значение `ptr` в функции является копией адреса массива, `ptr` всё равно указывает на фактический массив (а не на копию!). Следовательно, при разыменовании `ptr`, разыменовывается и фактический массив!
- Этот феномен работает так же и с указателями на значения не из массива.

Использование указателя для итерации по массиву

```
#include <iostream>
int main()
{
      const int arrayLength = 9;
      char name[arrayLength] = "Jonathan";
      int numVowels(0);
      for (char *ptr = name; ptr < name + arrayLength; ++ptr)
      {
            switch (*ptr)
            {
            case 'A':
            case 'a':
```

```
            case 'E':
            case 'e':
            case 'I':
            case 'i':
            case 'O':
            case 'o':
            case 'U':
            case 'u':
                  ++numVowels;
            }
      }

      std::cout << name << " has " << numVowels << " vowels.\n";
      return 0;
}
```

Результат выполнения программы:

```
Jonathan has 3 vowels.
```

Символьные константы строк C-style

Мы уже знаем, как создать и инициализировать строку `C-style`:

```
#include <iostream>
int main()
{
    char myName[] = "John";
    std::cout << myName;
    return 0;
}
```

Язык C++ поддерживает ещё один способ создания символьных констант строк `C-style` — через указатели:

```
#include <iostream>
int main()
{
    const char *myName = "John";
    std::cout << myName;
    return 0;
}
```

Хотя обе эти программы работают и выдают одинаковые результаты, выделение памяти в них выполняется по-разному.

- В первом случае в программе выделяется память для фиксированного массива длиной `5` и инициализируется эта память строкой `John\0`. Поскольку память была специально выделена для массива, то мы можем изменять её содержимое. Сам массив рассматривается как обычная локальная переменная, поэтому, когда он выходит из области видимости, память, используемая им, освобождается для других объектов.
- Во втором случае с символьной константой компилятор помещает строку `John\0` в память типа `read-only` (только чтение), а затем создаёт указатель, который указывает на эту строку. Несколько строковых литералов с одним и тем же содержимым могут указывать на один и тот же адрес. Поскольку эта память доступна только для чтения, а также потому, что внесение изменений в строковый литерал может повлиять на дальнейшее его использование, лучше всего перестраховаться, объявив строку константой (типа `const`). Также, поскольку строки, объявленные

таким образом, существуют на протяжении всей жизни программы (они имеют статическую продолжительность, а не автоматическую, как большинство других локально определённых литералов), нам не нужно беспокоиться о проблемах, связанных с областью видимости.

Поэтому следующее в порядке вещей:

```
const char* getName()
{
    return "John";
}
```

В фрагменте, приведенном выше, функция `getName()` возвращает указатель на строку `C-style John`. Всё хорошо, так как `John` не выходит из области видимости, когда `getName()` завершает своё выполнение, поэтому вызывающий объект всё равно имеет доступ к строке.

std::cout и указатели типа char

```
#include <iostream>
int main()
{
    int nArray[5] = { 9, 7, 5, 3, 1 };
    char cArray[] = "Hello!";
    const char *name = "John";
    std::cout << nArray << '\n'; // nArray распадается в указатель типа int
    std::cout << cArray << '\n'; // cArray распадается в указатель типа char
    std::cout << name << '\n'; // name уже и так является указателем типа char
    return 0;
}
```

Результат:

```
0x7ffdc0b6b620
Hello!
John
```

В массиве типа `int` выводится адрес, а в массивах типа `char` — строки!
Дело в том, что при передаче указателя не типа `char`, в результате выводится просто содержимое этого указателя (адрес памяти). Однако, если вы передадите объект типа `char*` или `const char*`, то `std::cout` предположит, что вы намереваетесь вывести строку. Следовательно, вместо вывода значения указателя — выведется строка, на которую тот указывает!
Это всё замечательно в 99% случаев, но это может привести и к неожиданным результатам, например:

```
#include <iostream>
int main()
{
    char a = 'R';
    std::cout << &a;
    return 0;
}
```

Здесь мы намереваемся вывести адрес переменной `a`. Тем не менее, `&a` имеет тип `char*`, поэтому `std::cout` выведет это как строку!
Результат выполнения программы может быть, например, таким:

R╠╠;¿■c

Почему так? `std::cout` предположил, что `&a` (типа `char*`) является строкой. Поэтому сначала вывелось `R`, а затем вывод продолжился. Следующим в памяти был мусор. В конце концов, `std::cout` столкнулся с ячейкой памяти, имеющей значение `0`, которое он интерпретировал как нуль-терминатор, и, соответственно, прекратил вывод. То, что вы видите в результате, может отличаться, в зависимости от того, что находится в памяти после переменной `a`.

Динамическое выделение памяти

Динамическое выделение переменных

Для динамического выделения памяти одной переменной используется оператор `new`:

```
new int; /* динамически выделяем целочисленную переменную и сразу же отбрасываем результат (так как нигде его не сохраняем) */
```

В примере, приведённом выше, мы запрашиваем выделение памяти для целочисленной переменной из операционной системы. Оператор `new` возвращает указатель, содержащий адрес выделенной памяти.
Для доступа к выделенной памяти создаётся указатель:

```
int *ptr = new int; /* динамически выделяем целочисленную переменную и присваиваем её адрес ptr, чтобы затем иметь доступ к ней */
```

Затем мы можем разыменовать указатель для получения значения:

```
*ptr = 8; // присваиваем значение 8 только что выделенной памяти
```

Инициализация памяти динамически выделенной переменной

При динамическом выделении памяти переменной можно инициализировать её посредством прямой инициализации или `uniform`-инициализации (в C++11):

```
int *ptr1 = new int (7); // используем прямую инициализацию
int *ptr2 = new int { 8 }; // используем uniform-инициализацию
```

Когда уже всё, что требовалось, выполнено с динамически выделенной переменной — нужно явно указать для C++ освободить эту память. Для переменных это выполняется с помощью оператора `delete`:

```
// Предположим, что ptr ранее уже был выделен с помощью оператора new
delete ptr; // возвращаем память, на которую указывал ptr, обратно в операционную систему
ptr = 0; // делаем ptr нулевым указателем (используйте nullptr вместо 0 в C++11)
```

Висячие указатели

Язык C++ не предоставляет никаких гарантий относительно того, что произойдёт с содержимым освобождённой памяти или со значением удаляемого указателя.

- Указатель, указывающий на освобождённую память, называется висячим указателем.
- Разыменование или удаление висячего указателя приведёт к неожиданным результатам.

Рассмотрим следующую программу:

```
#include <iostream>
int main()
{
    int *ptr = new int; // динамически выделяем целочисленную переменную
    *ptr = 8; // помещаем значение в выделенную ячейку памяти
    std::cout <<"1. \n";
    delete ptr; /* возвращаем память обратно в операционную систему, ptr теперь
является висячим указателем */
    std::cout <<"2. \n";
    std::cout << *ptr; /* разыменование висячего указателя приведёт к
неожиданным результатам */
    std::cout <<"\n3. \n";
    delete ptr; /* попытка освободить память снова приведёт к неожиданным
результатам также */
    std::cout <<"4. \n";
    return 0;
}
```

Результат:

```
free(): double free detected in tcache 2
1.
2.
1633729156
3.
```

Процесс освобождения памяти может также привести и к созданию нескольких висячих указателей. Рассмотрим следующий пример:

```
#include <iostream>
int main()
{
    int *ptr = new int; // динамически выделяем целочисленную переменную
    int *otherPtr = ptr; /* otherPtr теперь указывает на ту же самую выделенную
память, что и ptr */
    delete ptr; /* возвращаем память обратно в операционную систему. ptr и
otherPtr теперь висячие указатели */
    ptr = 0; // ptr теперь уже nullptr
    // Однако, otherPtr по-прежнему является висячим указателем!
    return 0;
}
```

Рекомендации:

- Во-первых, старайтесь избегать ситуаций, когда несколько указателей указывают на одну и ту же часть выделенной памяти. Если это невозможно, то выясните, какой указатель из всех «владеет» памятью (и отвечает за её удаление), а какие указатели просто получают доступ к ней.
- Во-вторых, если вы удаляете указатель, и он сразу же после этого не выходит из области видимости, то его нужно сделать нулевым, т.е. присвоить значение `0` (или `nullptr` в C++11).

Оператор new (std::nothrow)

При запросе памяти из операционной системы в редких случаях она может быть не выделена (т.е. её может и не быть в наличии).
По умолчанию, если оператор new не сработал, память не выделилась, то генерируется исключение `bad_alloc`. Если это исключение будет неправильно обработано, то программа

просто прекратит своё выполнение (произойдёт сбой) с ошибкой необработанного исключения.
Во многих случаях процесс генерации исключения оператором `new` (как и сбой программы) нежелателен, поэтому есть альтернативная форма оператора `new`, которая возвращает нулевой указатель, если память не может быть выделена. Нужно просто добавить константу `std::nothrow` между ключевым словом `new` и типом данных:

```
int *value = new (std::nothrow) int; /* указатель value станет нулевым, если динамическое выделение целочисленной переменной не выполнится */
```

Разыменовывать его также не рекомендуется, так как это приведёт к неожиданным результатам (скорее всего, к сбою в программе). Поэтому наилучшей практикой является проверка всех запросов выделения памяти на их успешность:

```
int *value = new (std::nothrow) int; /* запрос на выделение динамической памяти для целочисленного значения */
if (!value) /* обрабатываем случай, когда new возвращает null (т.е. память не выделяется) */
{
    // Обработка этого случая
    std::cout << "Could not allocate memory";
}
```

Утечка памяти

Динамически выделенная память не имеет области видимости, т.е. она остаётся выделенной до тех пор, пока не будет явно освобождена или пока ваша программа не завершит своё выполнение (и операционная система очистит все буфера памяти самостоятельно). Однако указатели, используемые для хранения динамически выделенных адресов памяти, следуют правилам области видимости обычных переменных. Это несоответствие может вызвать интересное поведение, например:

```
void doSomething()
{
    int *ptr = new int;
}
```

Здесь мы динамически выделяем целочисленную переменную, но никогда не освобождаем память через использование оператора `delete`. Поскольку указатели следуют всем тем же правилам, что и обычные переменные, то, когда функция завершит своё выполнение, `ptr` выйдет из области видимости. Поскольку `ptr` — это единственная переменная, хранящая адрес динамически выделенной целочисленной переменной, то, когда `ptr` уничтожится, больше не останется указателей на динамически выделенную память. Это означает, что программа «потеряет» адрес динамически выделенной памяти. И в результате эту динамически выделенную целочисленную переменную нельзя будет удалить.
Это называется утечкой памяти.

- Утечка памяти происходит, когда ваша программа теряет адрес некоторой динамически выделенной части памяти (например, переменной или массива), прежде чем вернуть её обратно в операционную систему.

Когда это происходит, то программа уже не может удалить эту динамически выделенную память, поскольку она больше не знает, где она находится. Операционная система также не может использовать эту память, поскольку считается, что она по-прежнему используется вашей программой.
Утечки памяти «съедают» свободную память во время выполнения программы, уменьшая количество доступной памяти не только для этой программы, но и для других программ

также. Программы с серьёзными проблемами с утечкой памяти могут «съесть» всю доступную память, в результате чего ваш компьютер будет медленнее работать или даже произойдёт сбой. Только после того, как выполнение вашей программы завершится, операционная система сможет очистить и вернуть всю память, которая «утекла».
Хотя утечка памяти может возникнуть и из-за того, что указатель выходит из области видимости, возможны и другие способы, которые могут привести к утечкам памяти.

- Например, если указателю, хранящему адрес динамически выделенной памяти, присвоить другое значение:

```
int value = 7;
int *ptr = new int; // выделяем память
ptr = &value; // старый адрес утерян - произойдет утечка памяти
```

- Это легко решается удалением указателя перед операцией переприсваивания:

```
int value = 7;
int *ptr = new int; // выделяем память
delete ptr; // возвращаем память обратно в операционную систему
ptr = &value; // переприсваиваем указателю адрес value
```

- Кроме того, утечка памяти также может произойти и через двойное выделение памяти:

```
int *ptr = new int;
ptr = new int; // старый адрес утерян - произойдёт утечка памяти
```

Динамические массивы

Для выделения динамического массива и работы с ним используются отдельные формы операторов `new` и `delete`: `new[]` и `delete[]`.

```
#include <iostream>

int main()
{
    std::cout << "Enter a positive integer: ";
    int length;
    std::cin >> length;
    int *array = new int[length]; /* используем оператор new[] для выделения
массива. Обратите внимание, переменная length не обязательно должна быть
константой! */
    std::cout << "I just allocated an array of integers of length " << length <<
'\n';
    array[0] = 7; // присваиваем элементу под индексом 0 значение 7
    delete[] array; /* используем оператор delete[] для освобождения выделенной
массиву памяти */
    array = 0; // используйте nullptr вместо 0 в C++11
    return 0;
}
```

Инициализация динамических массивов

Если вы хотите инициализировать динамический массив значением `0`, то всё довольно просто:

```
int *array = new int[length]();
```

Начиная с C++11, появилась возможность инициализации динамических массивов через списки инициализаторов:

```
int fixedArray[5] = { 9, 7, 5, 3, 1 }; // инициализируем фиксированный массив
int *array = new int[5] { 9, 7, 5, 3, 1 }; // инициализируем динамический массив
```

В C++11 фиксированные массивы также могут быть инициализированы с использованием `uniform`-инициализации:

```
int fixedArray[5] { 9, 7, 5, 3, 1 }; /* инициализируем фиксированный массив в C++11 */
char fixedArray[14] { "Hello, world!" }; /* инициализируем фиксированный массив в C++11 */
```

В C++11 нельзя инициализировать динамический массив символов строкой `C-style`:

```
char *array = new char[14] { "Hello, world!" }; // не работает в C++11
```

Вместо этого вы можете динамически выделить `std::string` (или выделить динамический массив символов, а затем с помощью функции `strcpy_s()` скопировать содержимое нужной строки в этот массив).
Динамические массивы должны быть объявлены с явным указанием их длины:

```
int fixedArray[] {1, 2, 3}; // ок: неявное указание длины фиксированного массива

int *dynamicArray1 = new int[] {1, 2, 3}; /* не ок: неявное указание длины динамического массива */

int *dynamicArray2 = new int[3] {1, 2, 3}; /* ок: явное указание длины динамического массива */
```

Строки C-style: ввод и хранение строк произвольной длины

Строка `C-style` — это простой массив символов, который завершает специальный символ нуль-терминатор `'\0'`.
Динамическое выделение массивов позволяет задавать их длину во время выделения. Однако C++ не предоставляет встроенный способ изменения длины массива, который уже был выделен. Но и это ограничение можно обойти, динамически выделив новый массив, скопировав все элементы из старого массива, а затем удалив старый массив.

Пример

```
#include <iostream>
#include <fstream>

int main()
{
    std::ifstream fi("in.txt"); // для чтения из файла при запуске через IDE
    std::ofstream fo("out.txt"); // для записи в файл при запуске через IDE

    char* str {nullptr};
    char ch;
    int count {1};
    str = new char[count];
    int i{0};
```

```
//    while ((ch = std::cin.get()) != std::cin.eof() && ch != '\n') // из
консоли
    while ((ch = fi.get()) != std::cin.eof() && ch != '\n')  // из файла
    {
      if (i < count)
      {
            str[i] = ch;
            i++;
      } else
      {
            char* oldstr {str};
            count *= 2;
            str = new char[count];
            for(int j {0}; j < count / 2; j++)
            {
                  str[j] = oldstr[j];
            }
            str[i] = ch;
            i++;
            delete[] oldstr;
      }
    }

    // Не забывайте, что ещё нужно добавить в конец '\0'

    std::cout << str << std::endl;
    fo << str;

    delete[] str;

    return 0;
}
```

const с указателями и ссылками

Рассмотрим указатель, который будет указывать на константную переменную.

```
const int value = 7; // value - это константа
int *ptr = &value; /* ошибка компиляции: невозможно конвертировать const int* в
int* */
*ptr = 8; // изменяем значение value на 8
```

Фрагмент кода, приведённый выше, не скомпилируется: мы не можем присвоить неконстантному указателю константную переменную.

Указатели на константные значения

Указатель на константное значение — это неконстантный указатель, который указывает на неизменяемое значение. Для объявления указателя на константное значение, используется ключевое слово `const` перед типом данных:

```
const int value = 7;
const int *ptr = &value; /* здесь всё ок: ptr - это неконстантный указатель,
который указывает на "const int" */
*ptr = 8; // нельзя, мы не можем изменить константное значение
```

В примере, приведённом выше, `ptr` указывает на константный целочисленный тип данных. Рассмотрим следующий пример:

```
int value = 7; // value - это не константа
```

```
const int *ptr = &value; // всё хорошо
```

- Указатель на константную переменную может указывать и на неконстантную переменную (как в случае с переменной `value` в примере, приведённом выше).
- Указатель на константную переменную обрабатывает переменную как константу при получении доступа к ней независимо от того, была ли эта переменная изначально определена как `const` или нет.

Следующее в порядке вещей:

```
int value = 7;
const int *ptr = &value; // ptr указывает на "const int"
value = 8; /* переменная value уже не константа, если к ней получают доступ через неконстантный идентификатор */
```

Но не следующее:

```
int value = 7;
const int *ptr = &value; // ptr указывает на "const int"
*ptr = 8; /* ptr обрабатывает value как константу, поэтому изменение значения переменной value через ptr не допускается */
```

- Указателю на константное значение, который сам при этом не является константным (он просто указывает на константное значение), можно присвоить и другое значение:

```
int value1 = 7;
const int *ptr = &value1; // ptr указывает на const int

int value2 = 8;
ptr = &value2; // хорошо, ptr теперь указывает на другой const int
```

- Указатели на константные значения в основном используются в параметрах функций (например, при передаче массива) для гарантии того, что функция случайно не изменит значение(я) переданного ей аргумента.

Константные указатели

- Константный указатель — это указатель, значение которого не может быть изменено после инициализации.
- Для объявления константного указателя используется ключевое слово `const` между звёздочкой и именем указателя:

```
int value = 7;
int *const ptr = &value;
```

- Подобно обычным константным переменным, константный указатель должен быть инициализирован значением при объявлении.
- *Он всегда будет указывать на один и тот же адрес.*

В вышеприведенном примере `ptr` всегда будет указывать на адрес `value` (до тех пор, пока указатель не выйдет из области видимости и не уничтожится):

```
int value1 = 7;
int value2 = 8;

int * const ptr = &value1; /* ок: константный указатель инициализирован адресом value1 */
ptr = &value2; /* не ок: после инициализации константный указатель не может быть изменен */
```

- Однако, поскольку переменная `value`, на которую указывает указатель, не является константой, то её значение можно изменить путём разыменования константного указателя:

```
int value = 7;
int *const ptr = &value; // ptr всегда будет указывать на value
*ptr = 8; // ок, так как ptr указывает на тип данных (неконстантный int)
```

Константные указатели на константные значения

Можно объявить константный указатель на константное значение, используя ключевое слово const как перед типом данных, так и перед именем указателя:

```
int value = 7;
const int *const ptr = &value;
```

Константный указатель на константное значение нельзя перенаправить указывать на другое значение также, как и значение, на которое он указывает, — нельзя изменить.

Ссылки

Ссылка — это тип переменной в языке C++, который работает как псевдоним другого объекта или значения.

```
#include <iostream>
int main()
{
	int value = 7; // обычная переменная
	int &ref = value; // ссылка на переменную value

	value = 8; // value теперь 8
	ref = 9; // value теперь 9
	std::cout << value << std::endl; // выведется 9
	++ref;
	std::cout << value << std::endl; // выведется 10

	return 0;
}
```

В примере, приведённом выше, объекты `ref` и `value` обрабатываются как одно целое. Использование оператора адреса с ссылкой приведёт к возврату адреса значения, на которое ссылается ссылка:

```
std::cout << &value;    // выведется 0x7ffef068d76c
std::cout << &ref;      // выведется 0x7ffef068d76c
```

l-value, r-value и инициализация ссылок

- `l-value` — это объект, который имеет определённый адрес памяти (например, переменная `x`) и сохраняется за пределами одного выражения.
- `r-value` — это временное значение без определённого адреса памяти и с областью видимости выражения (т.е. сохраняется в пределах одного выражения).
- В качестве `r-values` могут быть как результаты выражения (например, `2 + 3`), так и литералы.
- Ссылки должны быть инициализированы при создании:

```
int value = 7;
int &ref = value; // корректная ссылка: инициализирована переменной value
```

```
int &invalidRef; // некорректная ссылка: ссылка должна ссылаться на что-либо
```

- В отличие от указателей, которые могут содержать нулевое значение, ссылки нулевыми быть не могут.
- Ссылки на неконстантные значения могут быть инициализированы только неконстантными `l-values`. Они не могут быть инициализированы константными `l-values` или `r-values`:

```
int a = 7;
int &ref1 = a; // ок: a - это неконстантное l-value

const int b = 8;
int &ref2 = b; // не ок: b - это константное l-value

int &ref3 = 4; // не ок: 4 - это r-value
```

- После инициализации изменить объект, на который указывает ссылка — нельзя.

```
int value1 = 7;
int value2 = 8;

int &ref = value1; // ок: ref - теперь псевдоним для value1
ref = value2; /* присваиваем 8 (значение переменной value2) переменной value1.
Здесь НЕ изменяется объект, на который ссылается ссылка! */
```

Ссылки в качестве параметров в функциях

Ссылки чаще всего используются в качестве параметров в функциях. В этом контексте `ссылка-параметр` работает как псевдоним аргумента, а сам аргумент не копируется при передаче в параметр. Это в свою очередь улучшает производительность, если аргумент слишком большой или затратный для копирования. Мы говорили о том, что передача `аргумента-указателя` в функцию позволяет функции при разыменовании этого указателя напрямую изменять значение аргумента. Ссылки работают аналогично. Поскольку `ссылка-параметр` — это псевдоним аргумента, то функция, использующая `ссылку-параметр`, может изменять аргумент, переданный ей, также напрямую:

```
#include <iostream>
 // ref - это ссылка на переданный аргумент, а не копия аргумента
void changeN(int &ref)
{
	ref = 8;
}

int main()
{
	int x = 7;
	std::cout << x << '\t';
	changeN(x); /* этот аргумент не обязательно должен быть ссылкой */
	std::cout << x;
	return 0;
}
```

Результат выполнения программы:

```
7 8
```

Ссылки на константные значения

- Ссылки на константные значения часто называют просто «ссылки на константы» или «константные ссылки».
- Объявить ссылку на константное значение можно путём добавления ключевого слова `const` перед типом данных.

```
const int value = 7;
const int &ref = value; // ref - это ссылка на константную переменную value
```

- Ссылки на константные значения могут быть инициализированы неконстантными `l-values`, константными `l-values` и `r-values`.

```
int a = 7;
const int &ref1 = a; // ок: a - это неконстантное l-value
const int b = 9;
const int &ref2 = b; // ок: b - это константное l-value
const int &ref3 = 5; // ок: 5 - это r-value
```

- Как и в случае с указателями, константные ссылки также могут ссылаться и на неконстантные переменные.
- При доступе к значению через константную ссылку, это значение автоматически считается `const`, даже если исходная переменная таковой не является.

```
int value = 7;
const int &ref = value; // создаем константную ссылку на переменную value

value = 8; // ок: value - это не константа
ref = 9; // нельзя: ref - это константа
```

Ссылки r-values

Обычно `r-values` имеют область видимости выражения, что означает, что они уничтожаются в конце выражения, в котором созданы:

```
std::cout << 3 + 4; /* 3 + 4 вычисляется в r-value 7, которое уничтожается в
конце этого стейтмента */
```

Когда константная ссылка инициализируется значением `r-value`, время жизни `r-value` продлевается в соответствии со временем жизни ссылки:

```
int somefcn()
{
    const int &ref = 3 + 4; /* обычно результат 3 + 4 имеет область видимости
выражения и уничтожился бы в конце этого стейтмента, но, поскольку результат
выражения сейчас привязан к ссылке на константное значение, то мы можем
использовать его здесь*/
    std::cout << ref;
} /* и время жизни r-value продлевается до этой точки, когда константная ссылка
уничтожается */
```

Константные ссылки в качестве параметров функции

Ссылки, используемые в качестве параметров функции, также могут быть константными. Это позволяет получить доступ к аргументу без его копирования, гарантируя, что функция не изменит значение, на которое ссылается ссылка:

```
// ref - это константная ссылка на переданный аргумент, а не копия аргумента
void changeN(const int &ref)
{
      ref = 8; // нельзя: ref - это константа
}
```

- Ссылки на константные значения особенно полезны в качестве параметров функции из-за их универсальности.
- Константная ссылка в качестве параметра позволяет передавать неконстантный аргумент `l-value`, константный аргумент `l-value`, литерал или результат выражения.

```
#include <iostream>

void printIt(const int &a)
{
    std::cout << a;
}

int main()
{
    int x = 3;
    printIt(x); // неконстантное l-value

    const int y = 4;
    printIt(y); // константное l-value

    printIt(5); // литерал в качестве r-value

    printIt(3+y); // выражение в качестве r-value

    return 0;
}
```

- Во избежание ненужного, слишком затратного копирования аргументов они должны передаваться в функцию по (константной) ссылке, при условии, что не требуется их изменение.
- Фундаментальные типы данных должны передаваться по значению в случае, если функция не будет изменять их значений.

Двумерные массивы

Указатели на указатели

Указатель на указатель — указатель, который содержит адрес другого указателя. Обычный указатель типа `int` объявляется с использованием одной звёздочки:

```
int *ptr; // указатель типа int, одна звёздочка
```

Указатель на указатель типа `int` объявляется с использованием 2-х звёздочек:

```
int **ptrptr; // указатель на указатель типа int (две звёздочки)
```

Указатель на указатель работает подобно обычному указателю: вы можете его разыменовать для получения значения, на которое он указывает. И, поскольку этим значением является другой указатель, для получения исходного значения вам потребуется выполнить разыменование ещё раз. Их следует выполнять последовательно:

```
#include <iostream>
int main()
{
	int value = 7;
	int *ptr = &value;
	std::cout << *ptr << std::endl; // разыменовываем указатель, чтобы
получить значение типа int
	int **ptrptr = &ptr;
	std::cout << **ptrptr << std::endl;
	return 0;
}
```

Результат выполнения программы:

```
7
7
```

- Нельзя инициализировать указатель на указатель напрямую значением.

```
int value = 7;
int **ptrptr = &&value; // нельзя
int **ptrptr = nullptr; // используйте 0, если не поддерживается C++11
```

Двумерные динамически выделенные массивы

Выделим массив указателей, а затем переберём каждый элемент массива указателей и выделим динамический массив для каждого элемента этого массива. Тогда наш динамический двумерный массив — это динамический одномерный массив динамических одномерных массивов!

```
int **array = new int*[15]; // выделяем массив из 15 указателей типа int – это
наши строки
for (int count = 0; count < 15; ++count)
    array[count] = new int[7]; // а это наши столбцы
```

- Доступ к элементам массива выполняется как обычно:

```
array[8][3] = 4; // это то же самое, что и (array[8])[3] = 4;
```

- Можем создать массив треугольной формы:

```
int **array = new int*[15]; // выделяем массив из 15 указателей типа int – это
наши строки
for (int count = 0; count < 15; ++count)
    array[count] = new int[count+1]; // а это наши столбцы
```

- array[0] — это массив длиной 1
- а array[1] — массив длиной 2 и т.д.

Освобождения памяти такого динамически выделенного двумерного массива:

```
for (int count = 0; count < 15; ++count)
    delete[] array[count];
delete[] array; // это следует выполнять в конце
```

Двумерный массив в одномерном

Вместо следующего:

```
int **array = new int*[15]; // выделяем массив из 15 указателей типа int – это наши строки
for (int count = 0; count < 15; ++count)
    array[count] = new int[7]; // а это наши столбцы
// Делаем следующее:
int *array = new int[105]; // двумерный массив 15x7 "сплющенный" в одномерный массив
```

Простая математика используется для конвертации индексов строки и столбца прямоугольного двумерного массива в один индекс одномерного массива:

```
int getSingleIndex(int row, int col, int numberOfColumnsInArray)
{
     return (row * numberOfColumnsInArray) + col;
}
// Присваиваем array[9,4] значение 3, используя наш "сплющенный" массив
array[getSingleIndex(9, 4, 5)] = 3;
```

Указатели для сортировки массивов функцией std::sort()

Поскольку операция сортировки массивов очень распространена, то Стандартная библиотека C++ предоставляет встроенную функцию сортировки `std::sort()`. Она находится в заголовочном файле `algorithm` и вызывается следующим образом:

```
#include <iostream>
#include <algorithm> // для std::sort()

int main()
{
      const int length = 5;
      int array[length] = { 30, 50, 20, 10, 40 };
      std::sort(array, array+length); // array – в качестве итератора указатель

      for (int i=0; i < length; ++i)
            std::cout << array[i] << ' ';

      return 0;
}
```

Работа с файлами

Файловые потоки

Для работы с файлами в стандартной библиотеке определён заголовочный файл `fstream`, который определяет базовые типы для чтения и записи файлов. В частности, это:

- `ifstream`: для чтения с файла
- `ofstream`: для записи в файл
- `fstream`: совмещает запись и чтение

Для работы с данными типа `wchar_t`[15] для этих потоков определены двойники:

- `wifstream`
- `wofstream`
- `wfstream`

15 wchar_t: представляет расширенный символ. На Windows занимает в памяти 2 байта (16 бит), на Linux - 4 байта (32 бита). Может хранить любой значение из диапазона от 0 до 65 535 (при 2 байтах), либо от 0 до 4 294 967 295 (для 4 байт).

Открытие файла

При операциях с файлом вначале необходимо открыть файл с помощью функции `open()`. Данная функция имеет две версии:

- `open(путь)`
- `open(путь, режим)`

Для открытия файла в функцию необходимо передать путь к файлу в виде строки. И также можно указать режим открытия. Список доступных режимов открытия файла:

- `ios::in`: файл открывается для ввода (чтения). Может быть установлен только для объекта `ifstream` или `fstream`.
- `ios::out`: файл открывается для вывода (записи). При этом старые данные удаляются. Может быть установлен только для объекта ofstream или fstream.
- `ios::app`: файл открывается для дозаписи. Старые данные не удаляются.
- `ios::ate`: после открытия файла перемещает указатель в конец файла.
- `ios::trunc`: файл усекается при открытии. Может быть установлен, если также установлен режим `out`.
- `ios::binary`: файл открывается в бинарном режиме.

Если при открытии режим не указан, то по умолчанию для объектов `ofstream` применяется режим `ios::out`, а для объектов `ifstream` - режим ios::in. Для объектов `fstream` совмещаются режимы `ios::out` и `ios::in`.

```
std::ofstream out;          // поток для записи
out.open("hello1.txt"); // окрываем файл для записи

std::ofstream out2;
out2.open("hello2.txt", std::ios::app); // окрываем файл для дозаписи

std::ofstream out3;
out2.open("hello3.txt", std::ios::out | std::ios::trunc); // установка
нескольких режимов

std::ifstream in;       // поток для чтения
in.open("hello4.txt"); // открываем файл для чтения

std::fstream fs;        // поток для чтения-записи
fs.open("hello5.txt"); // открываем файл для чтения-записи
```

Однако в принципе необязательно использовать функцию `open` для открытия файла. В качестве альтернативы можно также использовать `конструктор` объектов-потоков и передавать в них путь к файлу и режим открытия:

```
fstream(путь)
fstream(путь, режим)
```

При вызове `конструктора`, в который передан путь к файлу, данный файл будет автоматически открываться:

```
std::ofstream out("hello.txt");
std::ifstream in("hello.txt");
std::fstream fs("hello.txt", std::ios::app);
```

В данном случае предполагается, что файл `"hello.txt"` располагается в той же папке, где и файл программы.
Вообще использование `конструкторов` для открытия потока является более предпочтительным, так как определение переменной, представляющей файловой поток, уже предполагает, что этот поток будет открыт для чтения или записи. А использование

конструктора избавит от ситуации, когда мы забудем открыть поток, но при этом начнём его использовать.
В процессе работы мы можем проверить, открыт ли файл с помощью функции is_open(). Если файл открыт, то она возвращает true:

```
std::ifstream in;       // поток для чтения
in.open("hello.txt"); // открываем файл для чтения

if (in.is_open()) // если файл открыт
{
    // использование файла
}
```

Закрытие файла

После завершения работы с файлом его следует закрыть с помощью функции close(). Также стоит отметить то, что при выходе объекта потока из области видимости, он удаляется, и у него автоматически вызывается функция close.

```
#include <iostream>
#include <fstream>

int main()
{
    std::ofstream out;          // поток для записи
    out.open("hello.txt"); // окрываем файл для записи
    out.close();            // закрываем файл
    std::ifstream in;       // поток для чтения
    in.open("hello.txt"); // окрываем файл для чтения
    in.close();             // закрываем файл
    std::fstream fs;        // поток для чтения-записи
    fs.open("hello.txt"); // окрываем файл для чтения-записи
    fs.close();             // закрываем файл
}
```

Чтение и запись текстовых файлов

Потоки для работы с текстовыми файлами представляют объекты, для которых не задан режим открытия ios::binary.

Запись в файл

Для записи в файл к объекту ofstream или fstream применяется оператор << (как и при выводе на консоль):

```
#include <iostream>
#include <fstream>

int main()
{
    std::ofstream out;          // поток для записи
    out.open("hello.txt");      // открываем файл для записи
    if (out.is_open())
    {
        out << "Hello World!" << std::endl;
    }
    out.close();
    std::cout << "File has been written" << std::endl;
}
```

Здесь предполагается, что файла "hello.txt" располагается в одной папке с файлом программы. Данный способ перезаписывает файл заново. Если надо дозаписать текст в конец файла, то для открытия файла нужно использовать режим ios::app:

```
std::ofstream out("hello.txt", std::ios::app);
if (out.is_open())
{
    out << "Welcome to C++" << std::endl;
}
out.close();
```

Чтение из файла

Если надо считать всю строку целиком или даже все строки из файла, то лучше использовать встроенную функцию getline(), которая принимает поток для чтения и переменную, в которую надо считать текст:

```
#include <iostream>
#include <fstream>
#include <string>     // для std::getline

int main()
{
    std::string line;

    std::ifstream in("hello.txt"); // открываем файл для чтения
    if (in.is_open())
    {
        while (std::getline(in, line))
        {
            std::cout << line << std::endl;
        }
    }
    in.close();     // закрываем файл
}
```

Также для чтения данных из файла для объектов ifstream и fstream может применяться оператор >> (также как и при чтении с консоли):

```
#include <iostream>
#include <fstream>
#include <vector>

struct Point
{
    Point(double x, double y): x{x}, y{y} {}
    double x;
    double y;
};

int main()
{
    std::vector<Point> points{ Point{0, 0}, Point{4, 5}, Point{-5, 7}};
    std::ofstream out("points.txt"); // открываем файл для записи
    if (out.is_open())
    {
        // записываем все объекты Point в файл
        for (const Point& point: points)
        {
            out << point.x << " " << point.y << std::endl;
```

```
        }
    }
    out.close();

    std::vector<Point> new_points;
    std::ifstream in("points.txt"); // открываем файл для чтения
    if (in.is_open())
    {
        double x, y;
        while (in >> x >> y)
        {
            new_points.push_back(Point{x, y});
        }
    }
    in.close();

    for (const Point& point: new_points)
    {
        std::cout << "Point X: " << point.x << "\tY: " << point.y << std::endl;
    }
}
```

Здесь вектор структур `Point` записывается в файл.

```
for (const Point& point: points)
{
    out << point.x << " " << point.y << std::endl;
}
```

При чем при записи значений переменных в файл они отделяются пробелом. В итоге будет создаваться файл в формате

```
0 0
4 5
-5 7
```

Используя оператор `>>`, можно считать последовательно данные в переменные `x` и `y` и ими инициализировать структуру.

```
double x, y;
while (in >> x >> y)
{
    new_points.push_back(Point{x, y});
}
```

Но стоит отметить, что это ограниченный способ, поскольку при чтении файла поток `in` использует пробел для отделения одного значения от другого и таким образом считывает эти значения в переменные `x` и `y`. Если же нам надо записать и затем считать строку, которая содержит пробелы, и какие-то другие данные, то такой способ, конечно, не сработает.

Практическая часть

1. Ознакомится с теоретическим текстовым материалом изложенным в пояснении к лабораторной работе.
2. Получить задание у преподавателя (два номера задач).
3. Обеспечить в разрабатываемых программах дружественный консольный интерфейс. Программа должна предлагать пользователю вводить исходные данные (с пояснением того, что нужно ввести), а также при выводе результат её работы должен сопровождаться пояснением (комментарием). Например, при вводе программа запрашивает число строк и столбцов матрицы (n = 2 `\n` m = 3 `\n` Введите через разделитель элементы матрицы `\n` 1 2 3 1 0 1 `\n`), а при выводе — номер строки с наибольшим элементом заданной матрицы (Наибольший элемент располагается в 1-ой строке).
4. Обеспечить в разрабатываемых программах защиту от некорректного пользовательского ввода. При этом следует продумать возможные случаи такого некорректного ввода. Перечислить в отчёте примеры возможных вариантов некорректного пользовательского ввода. Продемонстрировать то, каким образом разрабатываемые программы обрабатывают некорректный пользовательский ввод. Достаточно привести по одному примеру на один случай некорректного пользовательского ввода.
5. Обеспечить в разрабатываемых программах возможность многократного ввода исходных данных необходимых для решения поставленной задачи. Программа должна спрашивать примерно следующее: «Хотите повторить ввод исходных данных? Да — 1, Нет — 0.» Также в некотором виде должен формироваться запрос(ы), определяющие откуда поступят исходные данные и куда будет осуществлён вывод результата.
6. Обеспечить ввод исходных данных из файла / консоли. Смотри таблицу 4.1.
7. Обеспечить вывод результатов в файл / консоль. Смотри таблицу 4.1.
8. В случае ввода данных из файла программа должна завершаться (не предлагать повторно ввести исходные данные). Смотри таблицу 4.1.
9. Имя файла (путь к файлу) с исходными данными и к файлу для размещения результатов работы программы задать в коде программы в виде констант.
10. Для инициализации переменных использовать `uniform`-инициализацию.
11. Обеспечить работу со строками произвольной длины, то есть считать, что заранее не известно то, какой именно длины будет последовательность символов, задаваемая пользователем. Не следует обращать внимание на то, что в индивидуальном задании указывается количество символов входной последовательности. Считать такое указание один из многих вариантов размерности, заданной пользователем.
12. Обеспечить работу с матрицами произвольной размерности. Пользователь может задать любое количество строк и столбцов. Если в индивидуальном задании указывается какая-то конкретная размерность для матрицы, то считать такое указание один из многих вариантов размерности, заданной пользователем.

13. Для хранения данных использовать динамические массивы (запрещается использовать контейнеры типа `str::array` или `std::vector` или `std::string` и тому подобные, допускается только `Type* my_mas = new Type[N];`).
14. Для массивов использовать передачу в функции по адресу или по ссылке. Если функция не изменяет массив, то соответствующий параметр для передачи массива должен быть объявлен как `const`.
15. Выделение памяти должно сопровождаться процедурой проверки успешности выделения.
16. В программе не должно быть висячих указателей и ситуаций утечки памяти.
17. Для зануления указателей использовать значение `nullptr`.
18. Выполнить по заданию преподавателя одну указанную задачу из диапазона 251 — 270.[16] При выполнении задачи осуществить обработку последовательности символов, длина последовательности заранее неизвестна.
19. Выполнить по заданию преподавателя одну указанную задачу из диапазона 367 — 423. При выполнении задачи выполнить манипуляции с матрицей, используя вложенные циклы.
20. Задания с графикой давать не нужно.
21. Оформить программу как многофайловую.
    * Используемые константы расположить в отдельном заголовочном файле `*.h`.
    * Дополнительно поместить используемые константы в отдельное пространство имён.
    * Прототипы пользовательских функций расположить в отдельном заголовочном файле(файлах) `*.h`.
    * Пользовательские функций расположить в отдельном файле(файлах) `*.cpp`.
    * В заголовочных файлах использовать механизм `Header guards` или `#pragma once`.
    * Таким образом, в главном файле проекта должна располагаться только функция `main`, в которой подключается ряд пользовательских заголовочных файлов и осуществляется вызов пользовательских функций.
22. Оформить по выполненной лабораторной работе в рабочей тетради письменный отчёт.[17]

16 «Задачи по программированию». Авторы: С.А. Абрамов, Г.Г. Гнездилова, Е.Н. Капустина, М.И. Селюн. Компьютерный набор и оформление: Е.А. Гречникова. Лежит в ББ.

17 **Содержание отчёта:**
1. Задания на лабораторную работу.
2. Листинги программ.
3. Примеры выполнения программы (исходные данные + результаты).
4. Перечисление примеров возможных вариантов некорректного пользовательского ввода (начиная с лабораторной работы №3).

Примечание: Рабочая тетрадь с отчётами подписывается с указанием ФИО и номера учебной группы.

Табл. 4.1. Пояснение к пунктам 6 - 8

Входой поток	**Выходной поток**	**Завершение / Вопрос о продолжении**
Консоль	Консоль	Вопрос о продолжении
Консоль	Файл	Вопрос о продолжении
Файл	Консоль	Завершение
Файл	Файл	Завершение