**Федеральное государственное бюджетное образовательное учреждение**
**высшего образования**
**«МИРЭА – Российский технологический университет»**
**РТУ МИРЭА**
**Институт Информационных Технологий**
**Кафедра Промышленной Информатики**


# ПРОЦЕДУРНОЕ ПРОГРАММИРОВАНИЕ

## Тема лекции «Указатели. Динамические массивы»

**Лектор Каширская Елизавета Натановна** (к.т.н., доцент, ФГБОУ ВО "МИРЭА - Российский технологический университет") e-mail: liza.kashirskaya@gmail.com

**Тема № 5**

# УКАЗАТЕЛИ

При выполнении любой программы все необходимые для ее работы данные должны быть загружены в оперативную память компьютера. Для обращения к переменным, находящимся в памяти, используются специальные адреса, которые записываются в шестнадцатеричном виде.

Если переменных в памяти потребуется слишком большое количество, которое не сможет вместить в себя сама аппаратная часть, произойдет перегрузка системы или её зависание.

Если мы объявляем переменные статично, так как мы делали в предыдущих уроках, они остаются в памяти до того момента, как программа завершит свою работу, после чего уничтожаются.

Такой подход может быть приемлем в простых примерах и несложных программах, которые не требуют большого количества ресурсов. Если же наш проект является огромным программным комплексом с высоким функционалом, объявлять таким образом переменные, естественно, было бы довольно глупо.

Можете себе представить, если бы в компьютерных играх использовался такой метод работы с данными, геймерам пришлось бы перезагружать свои высоконагруженные системы после нескольких секунд игры.

Дело в том, что, играя в стратегию, геймер в каждый новый момент времени видит различные объекты на экране монитора, например, сейчас он стреляет во врага, а через долю секунды он уже падает убитым, создавая вокруг себя множество спецэффектов, таких как пыль, тени, и т.п.

Вы знаете, как на экране возникает иллюзия движения? Рисуется новое положение объекта и при этом старое стирается.

Естественно, все это занимает какое-то место в оперативной памяти компьютера. Если не стирать из памяти неиспользуемые объекты, очень скоро они заполнят весь объем ресурсов ПК.

На FullHD-экране (1920*1080 пикселей) и 24 бит на цвет каждого и с обновлениями 60 раз в секунду потребуется 356 Мб памяти каждую секунду. Поиграть можно будет примерно 10 секунд!

По этим причинам в большинстве языков, в том числе и C/C++, имеется понятие указателя. Указатель — это переменная, хранящая в себе адрес ячейки оперативной памяти.

Мы можем обращаться, например, к массиву данных через указатель, который будет содержать адрес начала диапазона ячеек памяти, хранящих этот массив.

После того, как этот массив станет не нужен для выполнения остальной части программы, мы просто освободим память по адресу этого указателя, и она вновь станет доступна для других переменных.

Ниже приведен конкретный пример обращения к переменным через указатель и напрямую.

## Пример использования статических переменных

```
#include <iostream>
using namespace std;

int main()
{
int a; // Объявление статической переменной
int b = 5; // Инициализация статической переменной b

    a = 10;
    b = a + b;
    cout << "b is " << b << endl;
    return 0;
}
```

## Пример использования динамических переменных

```
#include <iostream>
using namespace std;

int main()
{
    int *a = new int; // Объявление указателя для переменной типа int
    int *b = new int(5); // Инициализация указателя

    *a = 10;
    *b = *a + *b;

    cout << "b is " << *b << endl;

    delete b;
    delete a;

    return 0;
}
```

Синтаксис первого примера вам уже быть знаком. Мы объявляем/инициализируем статичные переменные **a** и **b**, после чего выполняем различные операции напрямую с ними.

Во втором примере мы оперируем динамическими переменными посредством указателей. Рассмотрим общий синтаксис указателей в C++.

**Выделение памяти** осуществляется с помощью оператора **new** и имеет вид: **тип_данных *имя_указателя = new тип_данных;**

*например*, **int *a = new int;**.

После удачного выполнения такой операции в оперативной памяти компьютера происходит выделение диапазона ячеек, необходимого для хранения переменной типа **int**.

Для разных типов данных выделяется разное количество памяти. Следует быть особенно осторожным при работе с памятью, потому что именно ошибки программы, вызванные утечкой памяти, являются одними из самых трудно находимых. На отладку программы в поисках одной ничтожной ошибки может уйти час, день, неделя, в зависимости от упорности разработчика и объема кода.

**Инициализация значения**, находящегося по адресу указателя, выполняется схожим образом, только в конце ставятся круглые скобки с нужным значением:

**тип данных *имя_указателя = new тип_данных(значение)**.

В нашем примере это **int *b = new int(5)**.

# Получение адреса

Для того, чтобы получить **адрес** в памяти, на который ссылается указатель, используется имя переменной-указателя с префиксом **&** перед ним (*не путать со знаком ссылки в C++*).

Например, чтобы вывести на экран адрес ячейки памяти, на который ссылается указатель **b** во втором примере, мы пишем cout << "Address of b is " << &b << endl;. В моей системе, я получила значение **0x1aba030**. У вас оно может быть другим, потому что адреса в оперативной памяти распределяются таким образом, чтобы максимально уменьшить фрагментацию. Поскольку в любой системе список запущенных процессов, а также объем и разрядность памяти могут отличаться, операционная система сама распределяет данные для обеспечения минимальной фрагментации.

# Пример получения адреса

```
cout << a << "!\n";
cout << &a << "!\n";
cout << a << "!\n";
cout << *a << "!\n";
```

У меня выводится так:
0x7f6c38d989c8!
0x3d88a00!
0x7f6c38d989c8!
0x3d88a00!

Адрес указателя не меняется.
**&a** - это адрес, хранимый в указателе.

А если при выводе написать просто b, он выведет адрес памяти?
Да. Указательная переменная хранит адрес смещения от начала.
При ***b** выведет значение, при **b** выведет адрес в HEX.

Для того, чтобы получить **значение**, которое находится **по адресу**, на который ссылается указатель, **используется префикс *.** Данная операция называется **разыменованием указателя**.

Во втором примере мы выводим на экран значение, *которое находится в ячейке памяти* (у меня это **0x1aba030**): cout << "b is " << *b << endl; . В этом случае необходимо использовать знак ***.

# Изменение значения

Чтобы изменить значение, находящееся по адресу, на который ссылается указатель, нужно также использовать звездочку, например, как во втором примере:

```
*b = *a + *b;
```

Когда мы оперируем **данными**, то используем знак ***

Когда мы оперируем **адресами**, то используем знак **&**

Для того, чтобы освободить память, выделенную оператором **new**, используют оператор delete.

# Пример освобождения памяти

```
#include <iostream>
using namespace std;

int main()
{
    // Выделение памяти
    int *a = new int;
    int *b = new int;
    float *c = new float;

    // ... Любые действия программы

    // Освобождение выделенной памяти
    delete c;
    delete b;
    delete a;

    return 0;
}
```

# Выделение памяти с помощью операции new

До сих пор мы инициализировали указатели адресами переменных; переменные — это именованная память, выделенная во время компиляции, и каждый указатель, до сих пор использованный в примерах, просто представлял собой псевдоним для памяти, доступ к которой и так был возможен по именам переменных. Реальная ценность указателей проявляется тогда, когда во время выполнения выделяются неименованные области памяти для хранения значений. В этом случае указатели становятся единственным способом доступа к такой памяти.

# Выделение памяти с помощью операции new

В языке С память можно выделять с помощью библиотечной функции **malloc** (). Ее можно применять и в С++, но язык С++ также предлагает лучший способ — **операцию new**.

Создадим неименованное хранилище во время выполнения программы для значения типа int и обеспечим к нему доступ через указатель. Ключом ко всему является операция **new**. Вы сообщаете операции **new**, для какого типа данных запрашивается память; операционная система находит блок памяти нужного размера и возвращает операции **new** его адрес. Вы присваиваете этот адрес указателю, и на этом все. Ниже показан пример:

```
int *pn = new int;
```

# Выделение памяти с помощью операции new

Правая часть **new int** сообщает программе, что требуется некоторое новое хранилище, подходящее для хранения int. Операция **new** использует тип для того, чтобы определить, сколько байт необходимо выделить. Затем этот размер сообщается операционной системе, и она находит память и возвращает адрес. Далее вы присваиваете адрес переменной pn, которая объявлена как указатель на int. Теперь **pn — адрес,** а ***pn — значение, хранящееся по этому адресу**. Сравните это с присваиванием адреса переменной указателю:

```
int higgens;
int *pt = &higgens;
```

В обоих случаях (pn и pt) **вы присваиваете адрес** значения int **указателю**. Во втором случае вы также можете обратиться к int по имени **higgens**. В первом случае доступ возможен только через указатель. Возникает вопрос: поскольку память, на которую указывает pn, не имеет имени, как обращаться к ней? Мы говорим, что **pn** указывает на объект данных. Это не "объект" в терминологии объектно-ориентированного программирования. Это просто объект, в смысле "вещь". Термин "объект данных" является более общим, чем "переменная", потому что он означает любой блок памяти, выделенный для элемента данных. Таким образом, переменная — это тоже объект, но память, на которую указывает pn, не является переменной. Метод обращения к объектам данных через указатель может показаться поначалу несколько запутанным, однако он обеспечивает программе высокую степень управления памятью.

# Выделение памяти с помощью операции new

Общая форма получения и назначения памяти отдельному объекту данных, который может быть как структурой, так и фундаментальным типом, выглядит следующим образом:

**имяТипа *имя_указателя = new имяТипа;**

**Тип данных используется дважды: один раз для указания разновидности запрашиваемой памяти, а второй — для объявления подходящего указателя.**

Разумеется, если вы уже ранее объявили указатель на конкретный тип, то можете его применить вместо объявления еще одного. В листинге демонстрируется применение **new** для двух разных типов. Естественно, точные значения адресов памяти будут варьироваться от системы к системе.

# Выделение памяти с помощью операции new

```
#include<iostream>
int main()
{
using namespace std;
int nights = 1001;
int * pt = new int; // выделение пространства для int
*pt = 1001; // сохранение в нем значения
cout << "nights value = "; // значение nights
cout << nights << " : location " << &nights << endl; // расположение nights
cout << "int "; // значение и расположение int
cout << "value = " << *pt << " : location = " << pt << endl;
double * pd = new double; // выделение пространства для double
*pd = 10000001.0; // сохранение в нем значения double
cout << "double ";
cout << "value = " << *pd << ": location = " << pd << endl;
// значениерасположение double
cout << "location of pointer pd: " <<&pd << endl;
// расположениеуказателя pd
cout << "size of pt = " << sizeof(pt);
cout << " : size of *pt = " << sizeof(*pt) << endl;
cout << "size of pd = " << sizeof pd;
cout << ": size of *pd = " << sizeof (*pd) << endl;
return 0;
}
```

Ниже показан вывод программы из листинга:

```
nights value = 1001 : location 0x7ffe7bdf0d1c
int value = 1001 : location = 0x2387c20
double value = 1e+07: location = 0x2387c40
location of pointer pd: 0x7ffe7bdf0d20
size of pt = 8 : size of *pt = 4
size of pd = 8: size of *pd = 8
```

Если все, что нужно программе — это единственное значение, вы можете объявить обычную переменную, поскольку это намного проще (хотя и не так впечатляет), чем применение **new** для управления единственным небольшим объектом данных. Использование операции **new** более типично с крупными фрагментами данных, такими как массивы, строки и структуры. Именно в таких случаях операция **new** является полезной.

# Зачем вообще нужны new и delite?

Операции new и delete в C++ нужны для создания и удаления динамических объектов.

Основная особенность динамических объектов в том, что временем их жизни нужно управлять вручную. Противоположность им с этой точки зрения составляют автоматические объекты, временем жизни которых управляет компилятор (статические объекты).
Автоматические объекты удаляются неявно в соответствии с чёткими правилами, которые реализованы в компиляторе. Локальные переменные функции удаляются, когда поток управления покидает область видимости, в которой они объявлены.
А вот для динамических объектов таких правил нет. Их нужно всегда удалять явно.

Динамические объекты в C++ создаются с помощью new, а удаляются с помощью delete. Вот отсюда и все проблемы: никто не говорил, что эти конструкции следует использовать напрямую! Это низкоуровневые вызовы, они как бы под капотом. И не нужно лезть под капот без необходимости.

# Зачем вообще нужны new и delite?

Операции new и delete в C++ нужны для создания и удаления динамических объектов.

Основная особенность динамических объектов в том, что временем их жизни нужно управлять вручную. Противоположность им с этой точки зрения составляют автоматические объекты, временем жизни которых управляет компилятор (статические объекты).
Автоматические объекты удаляются неявно в соответствии с чёткими правилами, которые реализованы в компиляторе. Локальные переменные функции удаляются, когда поток управления покидает область видимости, в которой они объявлены.
А вот для динамических объектов таких правил нет. Их нужно всегда удалять явно.

Динамические объекты в C++ создаются с помощью new, а удаляются с помощью delete. Вот отсюда и все проблемы: никто не говорил, что эти конструкции следует использовать напрямую! Это низкоуровневые вызовы, они как бы под капотом. И не нужно лезть под капот без необходимости.

# В чем проблема с new и delite?

С самого момента своего изобретения
операторы new и delete используются неоправданно часто. Самые большие проблемы относятся к оператору delete:
Можно вообще забыть вызвать delete (утечка памяти, memory leak).
Можно забыть вызвать delete в случае исключения или досрочного возврата из функции (тоже утечка памяти).
Можно вызвать delete дважды (двойное удаление, double delete).
Можно вызвать не ту форму оператора: delete вместо delete[] или наоборот (неопределённое поведение, undefined behavior).
Все эти ситуации приводят в лучшем случае к падениям программы, а в худшем к утечкам памяти.

# Использование операции new для создания динамических массивов

Если все, что нужно программе — это единственное значение, вы можете объявить обычную переменную, поскольку это намного проще (хотя и не так впечатляет), чем применение **new** для управления единственным небольшим объектом данных. Использование операции **new** более типично с крупными фрагментами данных, такими как массивы, строки и структуры. Именно в таких случаях операция **new** является полезной.

Динамические массивы

Динамический массив — это массив с элементами, выделенными в динамической памяти. Он необходим в случае, если размер неизвестен на этапе компиляции, или если размер достаточно большой, и мы не хотим выделять массив на стеке, размер которого обычно сильно ограничен.

# Динамические объекты

Динамические объекты обычно используются, когда невозможно привязать время жизни объекта к какой-то конкретной области видимости. Если это можно сделать, наверняка следует использовать автоматическую память. Но это предмет отдельной статьи.

Когда динамический объект создан, кто-то должен его удалить, и условно типы объектов можно разделить на две группы: те, которые никак не осведомлены о процессе своего удаления, и те, которые что-то подозревают. Будем говорить, что первые имеют стандартную модель управления памятью, а вторые — нестандартную.

К типам со стандартной моделью управления памятью относятся все стандартные типы.
Иными словами, тип со стандартной моделью управления памятью не предоставляет никаких дополнительных механизмов для управления своим временем жизни. Этим целиком и полностью должна заниматься пользовательская сторона.

# ДИНАМИЧЕСКИЕ МАССИВЫ

Поскольку массивам фиксированного размера память выделяется во время компиляции, то здесь мы имеем два ограничения:

Массивы фиксированного размера не могут иметь длину, основанную на любом пользовательском вводе или другом значении, которое вычисляется во время выполнения программы (runtime).

Фиксированные массивы имеют фиксированную длину, которую нельзя изменить.

Во многих случаях эти ограничения являются проблематичными. К счастью, C++ поддерживает еще один тип массивов, известный как **динамический массив**. Размер такого массива может быть установлен во время выполнения программы и его можно изменить.

# ДИНАМИЧЕСКИЕ МАССИВЫ

Мы уже разобрали понятие массива. При объявлении мы задавали массиву определенный постоянный размер. Возможно, кто-то пробовал делать так:

```
int n=10;
int arr[n];
```

Казалось бы, это должно сработать, ведь размер массива равен **n** только в момент инициализации массива, а при изменении n во время работы программы размер массива не изменится.

Но компилятор это не пропустит, так как int n=10; - это переменная, а нужна константа:

```
const int n=10;
```

# ДИНАМИЧЕСКИЕ МАССИВЫ

Как уже было сказано — при объявлении статического массива, его размером должна являться числовая константа, а не переменная. В большинстве случаев целесообразно выделять определенное количество памяти для массива, значение которого изначально неизвестно.

Например, необходимо создать динамический массив из **N** элементов, где значение **N** задается пользователем. Мы учились выделять память для переменных, используя указатели. Выделение памяти для динамического массива имеет аналогичный принцип.

# Использование операции new для создания динамических массивов

Предположим, например, что вы пишете программу, которой может понадобиться массив, а может, и нет — это зависит от информации, поступающий во время выполнения. Если вы создаете массив простым объявлением, пространство для него распределяется раз и навсегда — во время компиляции. Будет ли востребованным массив в программе или нет — он все равно существует и занимает место в памяти.

Пока что мы рассмотрим два важных обстоятельства относительно динамических массивов: как применять операцию new для создания массива и как использовать указатель для доступа к его элементам.

# Использование операции new для создания динамических массивов

Создать динамический массив на C++ легко: вы сообщаете операции **new** тип элементов массива и требуемое количество элементов. Синтаксис, необходимый для этого, предусматривает указание имени типа с количеством элементов в квадратных скобках. Например, если необходим массив из 10 элементов int, следует записать так:

**int *psome = new int [10] ;** // *получение блока памяти из 10 элементов типа int*

Операция new возвращает адрес первого элемента в блоке. В данном примере это значение присваивается указателю psome.

Как всегда, вы должны сбалансировать каждый вызов new соответствующим вызовом delete, когда программа завершает работу с этим блоком памяти. Однако использование new с квадратными скобками для создания массива требует применения альтернативной формы delete при освобождении массива:

**delete [] psome;** // *освобождение динамического массива*

Присутствие квадратных скобок сообщает операционной системе, что она должна освободить весь массив, а не только один элемент, на который указывает указатель. Обратите внимание, что скобки расположены между delete и указателем.

# Использование операции new для создания динамических массивов

Если вы используете new без скобок, то и соответствующая операция delete тоже должна быть без скобок. Если же new со скобками, то и соответствующая операция delete должна быть со скобками. Ранние версии C++ могут не распознавать нотацию с квадратными скобками. Согласно стандарту ANSI/ISO, однако, эффект от несоответствия формы new и delete не определен, т.е. вы не должны рассчитывать в этом случае на какое-то определенное поведение. Вот *пример*:

```
int *pt = new int;
short *ps = new short [500] ;
delete [] pt; // эффект не определен, не делайте так
delete ps; // эффект не определен, не делайте так
```

На самом деле, delete [] pt; не скомпилируется. При попытке компиляции вы получите:

delete[] applied to a pointer that was allocated with 'new'.

А если это исправить, то удаляется сам указатель.

# Правила использования операций new и delite для создания динамических массивов

• Не использовать delete для освобождения той памяти, которая не была выделена new.

• Не использовать delete для освобождения одного и того же блока памяти дважды.

• Использовать delete[], если применялась операция new[] для размещения массива.

• Использовать delete без скобок, если применялась операция new для размещения отдельного элемента.

• Помнить о том, что применение delete к нулевому указателю является безопасным (при этом ничего не происходит).

Теперь вернемся к динамическому массиву. Обратите внимание, что psome — это указатель на отдельное значение int, являющееся первым элементом блока. Отслеживать количество элементов в блоке возлагается на вас как разработчика. То есть, поскольку компилятор не знает о том, что psome указывает на первое из 10 целочисленных значений, вы должны писать свою программу так, чтобы она самостоятельно отслеживала количество элементов.

На самом деле программе, конечно же, известен объем выделенной памяти, так что она может корректно освободить ее позднее, когда вы воспользуетесь операцией delete []. Однако эта информация не является открытой; вы, например, не можете применить операцию sizeof, чтобы узнать количество байт в выделенном блоке.

# Использование операции new для создания динамических массивов

Общая форма выделения и назначения памяти для массива выглядит следующим образом:

```
имя_типа *имя_указателя = new имя_типа [количество_элементов];
```

Вызов операции new выделяет достаточно большой блок памяти, чтобы в нем поместилось количество_элементов типа имя_типа, и устанавливает в имя_указателя указатель на **первый** элемент. Как вы вскоре увидите, **имя_указателя можно использовать точно так же, как обычное имя массива.**

# Создание динамического массива

```
#include <iostream>
using namespace std;

int main()
{
    int num; // размер массива
    cout << "Enter integer value: ";
    cin >> num; // получение от пользователя размера массива

    int *p_darr = new int[num]; // Выделение памяти для массива
    for (int i = 0; i < num; i++) {
        // Заполнение массива и вывод значений его элементов
        p_darr[i] = i;
        cout << "Value of " << i << " element is " << p_darr[i] << endl;
    }
    delete [] p_darr; // очистка памяти
    return 0;
}
```

Синтаксис выделения памяти для массива имеет вид

**тип *указатель = new тип[размер]**

# Освобождение памяти с помощью операции delete

Использование операции new для запрашивания памяти, когда она нужна — одна из сторон пакета управления памятью C++. Второй стороной является операция delete, которая позволяет вернуть память в пул свободной памяти, когда работа с ней завершена. Это — важный шаг к максимально эффективному использованию памяти. Память, которую вы занимаете или освобождаете, затем может быть повторно использована другими частями программы. Операция delete применяется с указателем на блок памяти, который был выделен операцией new:

```
int *ps = new int;        // выделить память с помощью операции new
... // использовать память
delete ps; // по завершении освободить память с помощью операции delete
```

# Освобождение памяти с помощью операции delete

Освобождение памяти, на которую указывает указатель, не удаляет сам указатель. Вы можете повторно использовать тот же указатель — например, чтобы указать на другой выделенный new блок памяти. Вы всегда должны обеспечивать **сбалансированное применение new и delete**; в противном случае вы рискуете столкнуться с таким явлением, как утечка памяти, т.е. ситуацией, когда память выделена, но более не может быть использована. Если утечки памяти слишком велики, то попытка программы выделить очередной блок может привести к ее аварийному завершению.

# Освобождение памяти с помощью операции delete

Вы не должны пытаться освобождать блок памяти, который уже был однажды освобожден. Стандарт C++ гласит, что результат таких попыток не определен, а это значит, что последствия могут оказаться любыми. Кроме того, вы не можете с помощью операции delete освобождать память, которая была выделена посредством объявления обычных переменных:

```
int *ps = new int; // нормально
delete ps; // нормально
delete ps; // теперь - не нормально!
int jugs = 5; // нормально
int *pi = &jugs; // нормально
delete pi; // не допускается, память не была выделена new
```

# Освобождение памяти с помощью операции delete

**Обратите внимание, что обязательным условием применения операции delete является использование ее с памятью, выделенной операцией new.** Это не значит, что вы обязаны применять тот же указатель, который был использован с new - просто нужно задать тот же адрес:

```
int *ps = new int; // выделение памяти
int *pq = ps; // установка второго указателя на тот же блок
delete pq; // вызов delete для второго указателя
```

Обычно не стоит создавать два указателя на один и тот же блок памяти, т.к. это может привести к ошибочной попытке освобождения одного и того же блока дважды. Но применение второго указателя бывает оправдано, когда вы работаете с функциями, возвращающими указатель.

# Использование динамического массива

Как работать с динамическим массивом после его создания? Для начала подумаем о проблеме концептуально. Следующий оператор создает указатель psome, который указывает на первый элемент блока из 10 значений int:

int *psome = new int [10]; // *получить блок для 10 элементов типа int*

Представьте его как палец, указывающий на первый элемент. Предположим, что int занимает 4 байта. Перемещая палец на 4 байта в правильном направлении, вы можете указать на второй элемент. Всего имеется 10 элементов, что является допустимым диапазоном, в пределах которого можно передвигать палец. Таким образом, операция new снабжает компилятор всей необходимой информацией для идентификации каждого элемента в блоке.

Теперь взглянем на проблему практически. Как можно получить доступ к этим элементам? С первым элементом проблем нет. Поскольку psome указывает на первый элемент массива, то *psome и есть значение первого элемента. Но остается еще девять элементов. Простейший способ доступа к этим элементам может стать сюрпризом для вас, если вы не работали с языком С; просто используйте указатель, как если бы он был именем массива. То есть можно писать psome[0] вместо *psome[0] для первого элемента, psome[1] — для второго и т.д. Получается, что применять указатель для доступа к динамическому массиву очень просто, хотя не вполне понятно, почему этот метод работает. Причина в том, что С и С++ внутренне все равно работают с массивами через указатели.

# Использование динамического массива

Подобная эквивалентность указателей и массивов — одно из замечательных свойств C и C++. (Иногда это также и проблема, но это уже другая история.) В листинге ниже показано, как использовать new для создания динамического массива и доступа к его элементам с применением нотации обычного массива. В нем также отмечается фундаментальное различие между указателем и реальным именем массива.

```
#include <iostream>
using namespace std;
int main()
{
double * p3 = new double [3]; // пространство для 3 значений double
p3[0] = 0.2; // трактовать p3 как имя массива
p3[1] = 0.5 ;
p3[2] = 0.8 ;
cout <<"p3[1] is " << p3[1] <<".\n"; // вывод p3[1]
p3 = p3 + 1;  // увеличение указателя
cout << "Nowp3[0] is " <<p3[0] << " and  "; // выводp3[0]
cout << "p3[1] is " <<p3[1] << " .\n" ;  // выводp3[1]
p3 = p3 - 1;  // возврат указателя в начало
delete [] p3;  // освобождение памяти
return 0;
}
```

```
p3[1] is 0.5.
Nowp3[0] is 0.5 and  p3[1] is 0.8 .
```

# Использование динамического массива

Тут то же самое, но чуть более по-русски..

```
// arraynew.cpp -- использование операции new для массивов
#include <iostream>
int main()
{
   using namespace std;
   double * p3 = new double [3];   // пространство для 3 double
   p3[0] = 0.2;                    // трактовать p3 как имя массива
   p3[1] = 0.5;
   p3[2] = 0.8;
   cout << "p3[1] равно " << p3[1] << ".\n";
   p3 = p3 + 1;       // увеличить указатель
   cout << "Теперь p3[0] равно " << p3[0] << " и ";
   cout << "p3[1] равно " << p3[1] << ".\n";
   p3 = p3 - 1;       // вернуть указатель к началу
   delete [] p3;      // освободить память
   return 0;
}
```

Вывод этой программы выглядит следующим образом:

```
p3[1] равно 0.5.
Теперь p3[0] равно 0.5 и p3[1] равно 0.8.
```

Как видите, `arraynew.cpp` использует указатель `p3`, как если бы он был именем массива, с первым элементом `p3[0]` и так далее. Фундаментальное отличие между именем массива и указателем проявляется в следующей строке:

```
p3 = p3 + 1;       // допускается для указателей, но не для имен массивов
```

Вы не можете изменить значение для имени массива. Но указатель — переменная, а потому ее значение можно изменить. Обратите внимание на эффект от добавления 1 к p3. Теперь выражение p3[0] ссылается на бывший второй элемент массива. То есть добавление 1 к p3 заставляет p3 указывать на второй элемент вместо первого. Вычитание 1 из значения указателя возвращает его назад, в исходное значение, поэтому программа может применить delete[] с корректным адресом.

Этот пример показывает изменение способа доступа к информации.

Действительные адреса соседних элементов int отличаются на 2 или 4 байта, поэтому тот факт, что добавление 1 к p3 дает адрес следующего элемента, говорит о том, что арифметика указателей устроена специальным образом. Так оно и есть на самом деле.

**a[i] = *(a+i) = i[a]** - интересное следствие из арифметики указателей

# ВИДЫ МАССИВОВ В РАЗНЫХ ЯЗЫКАХ ПРОГРАММИРОВАНИЯ

Существует четыре вида массивов:
1) динамические массивы
2) массивы переменной длины
3) массив фиксированной длины
4) статический массив

Рассмотрим четыре массива более подробно.
1) **Динамический массив**
Для него должен существовать метод, который непосредственно меняет длину

2) **Массив переменной длины**
Массив переменной длины является частным случаем динамического массива. Для него тоже существует метод, который непосредственно меняет длину, но он более сложный

3) **Массив фиксированной длины**
Массив фиксированной длины является частным случаем массива переменной длины. В этом случае после присвоения значения длины мы больше не можем ее изменять.

4) **Статический массив**
Статический массив является частным случаем массива фиксированной длины.

# ДИНАМИЧЕСКИЙ МАССИВ И МАССИВ ПЕРМЕННОЙ ДЛИНЫ

В Википедии есть два отдельных листа об этом:

1) относительно динамических массивов: https://en.wikipedia.org/wiki/Dynamic_array

2) относительно массивов переменной длины: https://en.wikipedia.org/wiki/Variable-length_array

Массивы переменной длины имеют переменные **размеры, которые устанавливаются один раз** во время выполнения.

Динамические массивы также являются массивами переменной длины, но они **также могут изменять размер (re-dimension) после их создания** . Это позволяет массиву расти, чтобы вместить дополнительные элементы выше его первоначальной емкости. При использовании массива вам придется вручную изменить размер массива или перезаписать существующие данные.

# Массив переменной длины

**Динамическим** называется массив, размер которого может изменяться во время исполнения программы. Возможность изменения размера отличает динамический массив от статического, размер которого задаётся на момент компиляции программы. Для изменения размера динамического массива язык программирования, поддерживающий такие массивы, должен предоставлять встроенную функцию или оператор. Динамические массивы дают возможность более гибкой работы с данными, так как позволяют не прогнозировать хранимые объёмы данных, а регулировать размер массива в соответствии с реально необходимыми объёмами.

# Массив переменной длины

Также иногда к динамическим относят *массивы переменной длины*, размер которых не фиксируется при компиляции, а задаётся при создании или инициализации массива во время исполнения программы. От «настоящих» динамических массивов они отличаются тем, что для них не предоставляются средства автоматического изменения размера с сохранением содержимого, так что при необходимости программист должен реализовать такие средства самостоятельно.

**В самом языке Си** нет динамических массивов, но функции стандартной библиотеки malloc, free и realloc позволяют реализовать массив переменного размера:

```
int *mas = (int*)malloc(sizeof(int) * n); // Создание массива из n элементов типа int
... mas = (int*)realloc(mas, sizeof(int) * m); // Изменение размера массива с n на m с сохранением содержимого
... free(mas); // Освобождение памяти после использования массива
```

Неудобство данного подхода состоит в необходимости вычислять размеры выделяемой памяти, применять явное преобразование типа и тщательно отслеживать время жизни массива (как и всегда при работе с динамически выделенной памятью в Си).

Многомерный динамический массив может быть создан как массив указателей на массивы:

```
int **A = (int **)malloc(N*sizeof(int *));
for(int i = 0; i < N; i++)
{
A[i] = (int *)malloc(M*sizeof(int));
}
```

Однако рост размерности существенно усложняет процедуры создания массива и освобождения памяти по завершении его использования. Ещё более усложняется задача изменения размера массива по одной или нескольким координатам.

# Массив переменной длины

В C++ поддерживаются функции работы с памятью из стандартной библиотеки Си, но их использование не рекомендуется. Массив переменной длины здесь также можно выделить с помощью стандартных команд работы с динамической памятью new и delete:

```
// Создание массива длиной n
int *mas = new int[n];
...
// Освобождение памяти массива delete []mas;
```

Как и в случае с Си, здесь требуется отслеживать время жизни массива, чтобы избежать утечки памяти или, наоборот, преждевременного освобождения памяти. Аналога realloc здесь нет, так что изменить размер массива можно только вручную, выделив новую память нужного размера и перенеся в неё данные.

Библиотечным решением является шаблонный класс std::vector:

Приведем *пример* массива переменной длины:

```
void f(int dim)
 {
char str[dim]; /* символьный массив переменной длины */
/* ... */
}
```

Здесь размер массива str определяется значением переменной dim, которая передается в функцию f() как параметр. Таким образом, при каждом вызове f() создается массив str разной длины.

# Литература


1. Процедурное программирование на языках СИ и С++ : учебно-методическое пособие / Л. А. Скворцова [и др.]. — М.: РТУ МИРЭА, 2018. — 238 с. [Электронный ресурс]. Режим доступа: https://library.mirea.ru/books/53585
2. Трофимов В.В., Павловская Т.А. Алгоритмизация и программирование: учебник для академического бакалавриата. М.: Издательство Юрайт, 2017
3. [Электронный ресурс]. Режим доступа: https://www.intuit.ru/studies/courses/16740/1301/info
4. Уроки С++ с нуля. [Электронный ресурс]. Режим доступа: https://code-live.ru/tag/cpp-manual,
5. Введение в языки программирования Си С++. [Электронный ресурс]. Режим доступа: https://www.intuit.ru/studies/courses/1039/231/info