**МАССИВЫ**

**ОПРЕДЕЛЕНИЕ**

Массив является простейшим агрегатным типом. Он моделирует набор однотипных элементов, расположенных подряд в непрерывном отрезке памяти.

**ОБЪЯВЛЕНИЕ МАССИВОВ**

Если T некоторый тип, N константа или выражение, вычисляемое во время компиляции:

T a[N];

объявляет переменную a типа «массив из N элементов типа T» (array of N elements of the type T). Тип N должен иметь неявное приведение к типу std::size\_t, а его значение, называемое размером массива, должно быть больше нуля. Массив располагается в непрерывном отрезке памяти, под каждый элемент массива выделяется sizeof(T) байт, соответственно размер памяти, необходимой для размещения всего массива, равен N\*sizeof(T) байт. Эта величина ограничена сверху платформой и компилятором. Тип массива обозначается как T[N], то есть он включает тип элементов и размер массива. Таким образом, массивы, имеющие одинаковый тип элементов, но разный размер, будут иметь разный тип.

Такие массивы еще называют встроенными массивами (regular arrays), чтобы подчеркнуть отличие от других вариантов массивов, термин «массив» используется в программировании и в том числе в C++ достаточно широко.

Вот примеры правильных объявлений массивов:

**const** **int** N = 8;  
**constexpr** **int** Square(**int** n) { **return** n \* n; }  
  
**int** a1[1];  
**int** a2[N];  
**int** a3['Q'];  
**int** a4[Square(2)];

А вот примеры некорректных объявлений массивов:

**int** n;  
**int** b1[0]; // нулевой размер  
**int** b2[n]; // размер нельзя определить во время компиляции  
**int** b3["Q"]; // размер нельзя привести к size\_t

Доступ к элементам массива осуществляется через индексатор, значения индекса от 0 до N-1. Вот пример:

**int** a[4];  
a[0] = 42;  
**int** t = a[3];

Выход за границы массива не контролируется, ошибка может привести к неопределенному поведению.

В одной инструкции можно объявить несколько массивов, но размер должен быть указан для каждого.

**int** a[4], b[8];

Для типов массивов можно вводить псевдонимы. Можно использовать традиционный вариант с ключевым словом typedef:

**typedef** **int** I4[4];

или более современный (C++11) с ключевым словом using:

**using I4 = int[4];**

После этого массивы объявляются как простые переменные:

I4 a, b;

Это будет то же самое, что

**int** a[4], b[4];

**ОПЕРАТОРЫ И СТАНДАРТНЫЕ ФУНКЦИИ ДЛЯ РАБОТЫ С МАССИВАМИ**

Для работы с массивами можно использовать оператор sizeof и несколько стандартных функций и макросов.

Оператор sizeof возвращает полный размер массива в байтах, то есть размер элемента умноженный на размер массива.

Макрос \_countof() (в MSVS заголовочный файл <cstdlib>) возвращает размер массива, то есть количество элементов. В С++17 появился стандартный шаблон функции std::size(), которая делает то же самое (а еще имеет перегруженную версию, которая определяет размер стандартного контейнера).

**int** a[4];  
std::cout << **sizeof**(a) << ' ' << std::size(a) << '\n';

Вывод:

16 4

В C++11 в стандартной библиотеке появились свободные (не члены) шаблоны функций std::begin() и std::end(). Вызванная для массива std::begin() возвращает указатель на первый элемент массива, std::end() на past-the-last элемент. (Есть также константные версии: std::cbegin(), std::cend().) Это позволяет использовать массивы в диапазонном for.

**int** a[4]{ 4, 3, 2, 1 };  
  
**for** (**auto** t : a)  
{  
 std::cout << t << ' ';  
}

А также в стандартных алгоритмах:

std::sort(std::begin(a), std::end(a));

**РАЗМЕЩЕНИЕ В ПАМЯТИ**

Если массив объявлен статически, то есть в глобальной области видимости, в области видимости пространства имен или в качестве статического члена класса, то он размещается в статической памяти.

Массивам, объявленным локально, память выделяется на стеке. (Естественно, надо учитывать ограниченный размер стека при выборе размера локальных массивов.) Нестатические члены класса размещаются в границах экземпляра класса. Динамические массивы размещаются в динамической памяти.

**ОГРАНИЧЕНИЯ НА ТИПЫ ЭЛЕМЕНТОВ МАССИВОВ**

Нельзя объявить массив, элементы которого имеют тип void.

Нельзя объявить массив ссылок.  
  
**int** u, v;  
**int** &rr[2] = { u, v }; // ошибка

Вместо этого можно использовать массив константных указателей.

**int** **\*** **const** rr[2] = { &u, &v };

В C++11 появился шаблон std::reference\_wrapper<>. Он эмулирует интерфейс ссылки, но экземпляры конкретизации можно хранить в контейнерах и встроенных массивах. Но все же эмуляция интерфейса ссылки не совсем полная, иногда приходится использовать функцию-член get():

**int** u = 42, v = 5;  
std::reference\_wrapper<**int**> rr[2] = { u, v };  
std::cout << rr[0] << ' ' << rr[1] << '\n'; // вывод: 42 5  
++rr[0];  
rr[1].get() = 125; // get() необходим  
std::cout << u << ' ' << v << '\n'; // вывод: 43 125

Нельзя объявить массив функций.

**int** ff[2](**double**); // ошибка  
Вместо этого можно использовать массив указателей на функцию.

**int** (\*ff[2])(**double**);

Шаблон std::reference\_wrapper<> можно конкретизировать типом функции, но преимуществ перед указателем практически нет — функцию и так можно вызвать через указатель без разыменования, а инициализировать указатель именем функции без оператора &.

Массив нельзя объявить с помощью ключевого слова auto.

**auto** x[2] = {1, 2} // ошибка

**СВЕДЕНИЕ И КОПИРОВАНИЕ МАССИВОВ**

Размер массива является составной частью типа массива, но в определенных ситуациях она теряется и это делает тип массива в некотором смысле «неполноценным». Эта потеря называется сведение (decay, array-to-pointer decay). (Для перевода термина «decay» еще используется слово «низведение», также можно встретить «разложение».) Суть сведения заключается в том, что почти в любом контексте массив преобразуется к указателю на первый элемент и информация о размере теряется. Исключениями являются оператор sizeof, оператор & (взятия адреса) и инициализация ссылки на массив. Объявление с помощью ключевого слова decltype также правильно определяет тип массива, без сведения.

Конечно, тесную связь массивов и указателей отрицать нельзя. Вот стандартный (в стиле C) способ обработать все элементы массива:

**const** **int** N = 100;  
**int** a[N];  
**for** (**int** **\***d = a, \*end = d + N; d < end; ++d)  
{  
 \*d = rand();  
}

Но все же сведение можно отнести к «сишным архаизмам» и с ним надо быть внимательным и аккуратным, иначе можно столкнуться с не самыми приятными неожиданностями.

Вот как сведение влияет на объявления функций. Функции:

**void** Foo(**int** a[4]);  
**void** Foo(**int** a[]);  
**void** Foo(**int** **\***a);

не являются перегруженными функциями — это одно и то же. Размер надо передавать дополнительным параметром или использовать специальное соглашение для определения размера (например, завершающий ноль для строк).

При внешнем связывании массива также происходит сведение.  
  
// file 1  
**int** A[4];  
  
// file 2  
**extern** **int** A[];

Для размера также надо использовать дополнительную переменную или использовать специальное соглашение для определения размера.

При объявлении переменной с помощью ключевого слова auto также происходит сведение.

**int** a[4];  
**auto** da = a; // тип da выводится как int\*

При конкретизации шаблона функции:

**template**<**typename** T>

**void** Foo(T t);

Сведение вызывает дополнительные проблемы при использовании наследования. (в C ведь нет наследования.) Рассмотрим пример.

**class** B {/\* ... \*/};  
**class** D : **public** B {/\* ... \*/};  
**void** Foo(B[], **int** size); // обработка массива элементов типа B

Следующий код компилируется без ошибок и предупреждений.

D d[4];

Foo(d, \_countof(d));

Но если sizeof(B) < sizeof(D), то в теле Foo() смещение элементов массива d (кроме нулевого, конечно) будет определятся неправильно и, соответственно, почти всегда Foo() будет работать некорректно. Так что работать с массивами в полиморфном стиле, через указатель на базовый класс, нельзя.

**КОПИРОВАНИЕ**

Наряду со сведением (и тесно связанная с ним) есть еще одна особенность типа массива, которая делает его в некотором смысле «неполноценным». Массивы не поддерживают привычный синтаксис инициализации и присваивания, основанный на семантике копирования:

**using** I4 = **int**[4];  
I4 a;  
I4 b = a; // ошибка  
I4 b{a}; // ошибка  
I4 b(a); // ошибка  
I4 b2;  
b2 = a; // ошибка

Также функция не может возвращать массив.

I4 Foo(); // ошибка

Но если массив является членом класса/структуры/объединения, то копирующий конструктор и соответствующий оператор присваивания, генерируемые компилятором, выполняют поэлементное копирование такого массива.

**struct** X   
{   
 **int** A[4];   
};  
  
X x = { {1, 2, 3, 4} };  
X x2 = x; // копирование  
X x3{x}; // копирование  
X x4;  
x4 = x; // присваивание  
  
X Foo() // возвращаемое значение функции  
{  
 **return** { {4, 3, 2, 1} };  
}

Еще одна ситуация, когда происходит копирование массива — это захват массива по значению в лямбда-выражении.

**intptr\_t** x[4];  
**auto** f = [x]() { **return** **sizeof**(x) / **sizeof**(x[0]); };  
std::cout << f() << '\n';

Вывод:  
4

Но если используется инициализирующий захват (C++14), то происходит сведение.

**intptr\_t** u[4];  
**auto** g = [x = u]() { **return** **sizeof**(x) / **sizeof**(x[0]); };  
std::cout << g() << '\n';

Вывод:  
1

**ИНИЦИАЛИЗАЦИЯ МАССИВОВ**

Для описания правил инициализации массивов необходимо кратко рассказать о тривиальных типах.

**ТРИВИАЛЬНЫЕ ТИПЫ И НЕИНИЦИАЛИЗИРОВАННЫЕ ПЕРЕМЕННЫЕ**

Конструкторы и деструкторы можно назвать ключевыми элементами объектной модели С++. При создании объекта обязательно вызывается конструктор, а при удалении — деструктор. Но проблемы совместимости с С вынудили сделать некоторое исключение, и это исключение называется тривиальные типы. Они введены для моделирования сишных типов и сишного жизненного цикла переменных, без обязательного вызова конструктора и деструктора. Сишный код, если он компилируется и выполняется в С++, должен работать так же, как в С. К тривиальным типам относятся числовые типы, указатели, перечисления, а также классы, структуры, объединения и массивы, состоящие из тривиальных типов. Классы и структуры должны удовлетворять некоторым дополнительным условиям: отсутствие пользовательского конструктора, деструктора, копирования, присваивания, виртуальных функций.

Переменная тривиального типа будет неинициализированной, если не использовать какой-нибудь вариант явной инициализации. Для тривиального класса компилятор может сгенерировать конструктор по умолчанию и деструктор. Конструктор по умолчанию обнуляет объект, деструктор ничего не делает. Но этот конструктор будет сгенерирован и использован только, если использовать какой-нибудь вариант явной инициализации, иначе переменная останется неинициализированной.

Неинициализированная переменная устроена следующим образом: если она объявлена в области видимости пространства имен (глобально), будет иметь все биты нулевыми, если локально, или создана динамически, то получит случайный набор битов. Понятно, что использование такой переменной может привести к непредсказуемому поведению программы. Массивы достаточно часто имеют тривиальный тип и поэтому эта проблема для них весьма актуальна.

Неинициализированные константы тривиального типа выявляет компилятор, иногда он выявляет и другие неинициализированные переменные, но с этой задачей лучше справляются статические анализаторы кода.

В стандартной библиотеке С++11 есть шаблоны, называемые свойствами типов (заголовочный файл <type\_traits>).

Один из них позволяет определить, является ли тип тривиальным. Выражение std::is\_trivial<Т>::value имеет значение true, если T тривиальный тип и false в противном случае.

## **СИНТАКСИС ИНИЦИАЛИЗАЦИИ МАССИВОВ**

### **ОБЩИЕ ПОЛОЖЕНИЯ**

Если не использовать явную инициализацию, то для массивов нетривиального типа гарантируется вызов конструктора по умолчанию для каждого элемента. Естественно, что в этом случае такой конструктор должен быть, иначе возникает ошибка. Но для массивов тривиального типа или, если конструктор по умолчанию отсутствует или не устраивает, необходимо использовать явную инициализацию.

Со времен C массивы можно было инициализировать с помощью синтаксиса агрегатной инициализации:

**int** a[4] = { 1, 2, 3, 4 };

В С++11 появилась универсальная инициализация (uniform initialization) и теперь можно инициализировать так:

**int** a[4]{ 1, 2, 3, 4 };

Для универсальной инициализации также можно использовать =, и различать эти два типа инициализации не всегда просто, а, скорее всего, не очень нужно.

Размер массива можно не указывать, тогда он определится по числу инициализаторов.

**int** a[] { 1, 2, 3, 4 };

Если размер массива указан, то число инициализаторов не должно быть больше размера массива. Если размер массива больше числа инициализаторов, то для оставшихся элементов гарантируется вызов конструктора по умолчанию (который, естественно, должен быть), в том числе и для тривиальных типов. Таким образам, указав пустой список инициализации, мы гарантируем вызов конструктора по умолчанию для всех элементов массива тривиального типа.

**int** a[4]{};

Массивы констант тривиального типа требуют обязательного списка инициализации.

**const** **int** a[4] = { 3, 2, 1 };

Число инициализаторов может быть меньше размера массива, в этом случае оставшиеся элементы инициализируются конструктором по умолчанию.

Символьные массивы можно инициализировать строковым литералом.

**const** **char** str[] = "meow";  
**const** **wchar\_t** wstr[] = L"meow";

Размер такого массива будет на единицу больше числа символов строки, нужно хранить завершающий нулевой символ.

### **ИНИЦИАЛИЗАЦИЯ ЧЛЕНОВ КЛАССА**

В С++11 появилась возможность инициализировать массивы, являющиеся нестатическими членами класса. Это можно сделать двумя способами: непосредственно при объявлении или в списке инициализации членов при определении конструктора.

**class** X  
{  
 **int** a[4]{ 1, 2, 3, 4 };  
 **int** b[2];  
// ...  
**public**:  
 X(**int** u, **int** v) : b{ u, v }   
 {}  
// ...  
};

Правда в этом случае надо всегда явно задавать размер массива, неявное определение размера через список инициализации не разрешается.

Статические массивы, как и ранее, можно инициализировать только при определении, размер массива может быть определен через список инициализации.

**class** X  
{  
 **static** **int** A[];  
// ...  
};  
  
**int** X::A[] = { 1, 2, 3, 4 };

В C++17 появилась возможность объявлять статические члены (включая массивы) как inline. Таки члены можно инициализировать при объявлении, определение не обязательно.

**class** X  
{  
 **inline** **static** **int** A[]{ 1, 2, 3, 4 };  
// ...  
};

**ТРЕБОВАНИЯ К ИНИЦИАЛИЗАТОРАМ**

Выражения, стоящие в списке инициализации, вычисляются непосредственно перед инициализацией, они не обязаны быть известными на стадии компиляции (конечно, за исключением массивов, объявленных как constexpr). Требования к элементам списка инициализации такие же, как и к аргументу функции, имеющей параметр того же типа, что и элемент массива — должно существовать неявное преобразование от типа элемента списка инициализации к типу элемента массива. Пусть у нас есть объявление массива:

T a[] = {x1 /\*, ... \*/};

или

T a[]{x1 /\*, ... \*/};

Наличие нужного преобразования эквивалентно корректности инструкции

T t = x1;

Элемент списка инициализации может быть сам списком инициализации. В этом случае корректность этой инструкции также гарантирует корректную инициализацию элемента массива.

Рассмотрим пример.

**class** Int  
{  
 **int** m\_Value;  
**public**:  
 Int(**int** v) : m\_Value(v) {}  
// ...  
};  
// ...  
**int** x, y;  
// ...  
Int rr[] = { x, y };

Если мы объявим конструктор Int как explicit, то последнее объявление станет некорректным. В этом случае придется писать

Int rr[] = { Int(x), Int(y) };

Этот пример также демонстрирует как с помощью списка инициализации мы можем создать массив для типа, у которого нет конструктора по умолчанию. Но в этом случае число инициализаторов должно совпадать с размером массива.

# **УКАЗАТЕЛИ И ССЫЛКИ НА МАССИВЫ**

## **УКАЗАТЕЛИ НА МАССИВЫ**

Пусть у нас объявлен массив

T a[N];

Указатель на этот массив объявляется и инициализируется следующим образом:

T(\*pa)[N] = &a;

Для получения указателя используется традиционный оператор &. Тип указателя на массива обозначается как T(\*)[N].

Обратим внимание на использование скобок, без них мы получим объявление массива из N элементов типа указатель на T.

Указатель на массив — это не указатель на первый элемент (хотя побитово они, конечно, совпадают), здесь нет никакого сведения. Это полноценный тип, который «знает» размер массива. Поэтому при инициализации размеры должны совпадать.

**int** a[4];  
**int**(\*pa)[4] = &a; // OK  
**int**(\*p2)[2] = &a; // ошибка, размеры не совпадают

При инкременте указатель на массив увеличивается на размер всего массива, а не на размер элемента.

Для доступа к элементу массива через указатель надо использовать оператор \* и индексатор.

(\*pa)[3] = 42;

При использовании псевдонимов можно получить более привычный синтаксис объявления указателя на массив.

**using** I4 = **int**[4];  
I4 a{ 1, 2, 3, 4 };  
I4 \*pa = &a;

Также можно использовать auto, компилятор правильно выводит тип переменной как указатель на массив исходя из типа инициализатора.

**int** a[4];  
**auto** pa = &a; // тип pa выводится как int(\*)[4]

Понимание указателей на массивы необходимо для правильной работы с многомерными массивами, которые подробно будут рассмотрены далее.

**ССЫЛКИ НА МАССИВЫ**

Пусть у нас объявлен массив  
  
T a[N];

Ссылка на этот массив объявляется и инициализируется следующим образом:

T(&ra)[N] = a;

Как и для любой ссылки, инициализация переменной типа ссылка на массив является обязательной. Тип ссылки на массива обозначается как T(&)[N].

Также ссылку на массив можно инициализировать разыменованным указателем на массив.

T(\*pa)[N] = &a;  
T(&ra)[N] = \*pa;

Как и указатель, ссылка «знает» размер массива. Поэтому при инициализации размеры должны совпадать.

**int** a[4];  
**int**(&ra)[4] = a; // OK  
**int**(&r2)[2] = a; // ошибка, размеры не совпадают

Доступ к элементу массива через ссылку осуществляется так же, как и через идентификатор массива.

ra[3] = 0;

Ссылки на массивы как раз и являются теми средствами, с помощью которых можно обойти сведение.

Функция **void** Foo(T(&a)[N]); ожидает аргументы типа T[N], указатели для нее не подходят.

При использовании псевдонимов можно получить более привычный синтаксис объявления ссылки на массив.

**using** I4 = **int**[4];  
I4 a{ 1, 2, 3, 4 };  
I4 &ra = a;

Также можно использовать auto, компилятор выводит тип переменной как ссылка на массив.

**int** a[4];  
**auto** &ra = a; // тип ra выводится как int(&)[4]

Обратите внимание на наличие & после auto, без него произошло бы сведение, и тип ra вывелся бы как int\*.

При конкретизации шаблона функции

**template**<**typename** T>  
**void** Foo(T& t);

тип параметра шаблонной функции также будет выведен как ссылка на массив, если аргумент является массивом.

Особенно удобно использовать шаблоны с выводом типа и размера массива.

**template**<**typename** T, **std::size\_t** N>  
**void** Foo(T(&a)[N]);

При конкретизации такого шаблона компилятор выводит тип элементов T и размер массива N (который гарантировано больше нуля).

В качестве аргументов можно использовать только массивы, указатели будут отвергнуты. Именно этот прием используется при реализации макроса \_countof() и шаблона функции std::size(), а так же шаблонов функций std::begin() и std::end(), которые обеспечивают для массивов реализацию диапазонного for и делают более комфортной работу с алгоритмами.

# **МНОГОМЕРНЫЕ МАССИВЫ**

C++ не поддерживает настоящие многомерные массивы, то есть выражение a[N, M] некорректно, но многомерность моделируется в виде «массива массивов», то есть можно использовать выражение a[N][M].

Если T некоторый тип, N и M выражения, допустимые для определения размера массива, то инструкция.

T a[N][M];

объявляет a как массив массивов, массив из N элементов, каждый из которых является массивом из M элементов типа T. Такой массив будем называть двумерным массивом. Выражение a[i][j], где i от 0 до N-1, j от 0 до M-1, дает доступ к элементам этого массива. Первый индекс выбирает массив из массива массивов, второй выбирает элемент в этом массиве. Значение N можно назвать внешним размером двумерного массива, M внутренним. Тип многомерного массива обозначается как T[N][M].

Выражение a[i] является массивом из M элементов типа T. Соответственно к нему может быть применено сведение, у него можно взять адрес или использовать для инициализации ссылки.

T \*dai = a[i];  
T(\*pai)[M] = &a[i];  
T(&rai)[M] = a[i];

Сведение преобразует массив к указателю на элемент. Для двумерного массива этот элемент сам является массивом, а значит двумерный массив сводится к указателю на массив.

T a[N][M];  
T(\*da)[M] = a;

Таким образом, при передаче двумерного массива в функцию следующие варианты объявления соответствующего параметра эквивалентны:

**void** Foo(T a[N][M]);  
**void** Foo(T a[][M]);  
**void** Foo(T(\*a)[M]);

Это означает, что внешний размер двумерного массива теряется и его надо передавать отдельным параметром.

При использовании псевдонимов можно получить более лаконичный синтаксис объявления двумерных массивов.

**using** I4 = **int**[4];  
I4 b[2];

Это то же самое, что  
  
**int** b[2][4];

Двумерные массивы инициализируются следующим образом:  
  
**int** b[2][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};

Если нужно гарантировать только инициализацию по умолчанию, то можно использовать пустой список инициализации {}. Определения размера по списку инициализации возможно только по внешнему размеру.

**int** b[][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}}; // ОК  
**int** b[][] = {{1, 2, 3, 4}, {5, 6, 7, 8}}; // ошибка

Можно получить указатель на двумерный массив:  
  
T a[N][M];  
T(\*pa)[N][M] = &a;

Также можно получить ссылку. Вот пример использования ссылки на двумерный массив.

**template**<**typename** T, **std::size\_t** N, **std::size\_t** M>  
**void** Print2dArray(T(&a)[N][M])  
{  
 **for** (**int** i = 0; i < N; ++i)  
 {  
 **for** (**int** j = 0; j < M; ++j)  
 {  
 std::cout << a[i][j] << ' ';  
 }  
 std::cout << '\n';  
 }  
}  
// ...  
**int** b[2][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}};  
Print2dArray(b);

Вывод:

1 2 3 4

5 6 7 8

Двумерный массив хорошо согласуется с математическими матрицами. В объявлении

T mtx[N][M];

N можно интерпретировать как число строк матрицы, M как число столбцов, тогда mtx[i][j] это элемент матрицы, находящийся на пересечении i-й строки и j-го столбца, а mtx[i] это массив размера M, который представляет i-ю строку матрицы. Соответственно, такая матрица располагается в памяти по строкам. Правда в математике принято нумеровать строки и столбцы с единицы, а не с нуля.

# **ДИНАМИЧЕСКИЕ МАССИВЫ**

В C++ отсутствует тип «динамический массив». Имеются только операторы для создания и удаления динамического массива, доступ к нему осуществляется через указатели на начало массива (своего рода полное сведение). Размер такого массива надо хранить отдельно. Динамические массивы желательно инкапсулировать в C++ классы.

## **СОЗДАНИЕ И УДАЛЕНИЕ ДИНАМИЧЕСКОГО МАССИВА**

Если T некоторый тип, n переменная, значение которой может определяются в процессе выполнения программы, то инструкция

T \*pa = **new** T[n];

создает массив в динамической памяти. Тип переменной n должен приводиться к std::size\_t, значение может быть нулем. Размер памяти, необходимой для размещения массива, то есть n\*sizeof(T), ограничен сверху платформой и компилятором. Переменная pa указывает на первый элемент массива.

Если тип T тривиальный, то элементы будут иметь случайное значение, в противном случае для инициализации элементов будет использован конструктор по умолчанию.

В C++11 появилась возможность использовать список инициализации.

**int** **\***pa = **new** **int**[n]{1, 2, 3, 4};

Если число инициализаторов больше размера массива, то лишние не используются (компилятор может выдать ошибку, если значение n известно на стадии компиляции). Если размер массива больше числа инициализаторов, то для оставшихся элементов гарантируется вызов конструктора по умолчанию, в том числе и для тривиальных типов. Таким образам, указав пустой список инициализации, мы гарантируем вызов конструктора по умолчанию для всех элементов массива тривиального типа.

Оператор new[] сначала выделяет память для всего массива. Если выделение прошло успешно, то, если T нетривиальный тип или есть список инициализации, вызывается конструктор для каждого элемента массива начиная с нулевого. Если какой-нибудь конструктор выбрасывает исключение, то для всех созданных элементов массива вызывается деструктор в порядке, обратном вызову конструктора, затем выделенная память освобождается. Стандартные функции выделения памяти при невозможности удовлетворить запрос выбрасывают исключение типа std::bad\_alloc.

Динамический массив удаляется оператором delete[], который применяется к указателю, возвращаемому оператором new[].

**delete**[] pa;

При этом, если при создании массива использовался конструктор, то для всех элементов массива вызывается деструктор в порядке, обратном вызову конструктора (деструктор не должен выбрасывать исключений), затем выделенная память освобождается.

В остальных отношениях указатель pa, возвращаемый оператором new[], является просто указателем на начало массива, через него нельзя (во всяком случае «законно») получить размер массива, этот размер надо хранить отдельно. Соответственно с динамическим массивом нельзя использовать диапазонный for. Указатели в C/C++ поддерживают индексатор (встроенный оператор []), поэтому доступ к элементам динамического массива выглядит так же, как и к обычному массиву, контроля за корректностью индекса нет.

## **ДИНАМИЧЕСКИЕ МАССИВЫ И ИНТЕЛЛЕКТУАЛЬНЫЕ УКАЗАТЕЛИ**

Стандартный интеллектуальный указатель std::unique\_ptr<> можно использовать для управления жизненным циклом динамического массива (см. [Josuttis]). Он имеет частичную специализацию для массивов (см. раздел 7), которая перегружает оператор [] вместо операторов -> и \*, а также использует оператор delete[] в качестве удалителя по умолчанию. Вот пример:

**int** n = 100;  
std::unique\_ptr<**int**[]> aptr(**new** **int**[n]);  
**for** (**int** i = 0; i < n; ++i)  
{  
 aptr[i] = i;   
}

Эта поддержка не является полноценной: не хранится информация о размере массива, поэтому нет возможности контролировать корректностью индекса, не поддерживается интерфейс стандартных контейнеров и диапазонный for.

В C++14 появилась возможность создать динамический массив и инициализировать им экземпляр std::unique\_ptr <>  с помощью std::make\_unique<>:

**auto** aptr = std::make\_unique<**int**[]>(n);

При этом гарантируется инициализация элементов массива по умолчанию, в том числе и для тривиальных типов.

Интеллектуальный указатель std::shared\_ptr<> стал поддерживать такую специализацию только в C++17, а использование std::make\_shared<> для этой специализации появилось только в C++20.

В качестве альтернативы такому использованию интеллектуальных указателей можно рекомендовать std::vector<>.

**МНОГОМЕРНЫЕ ДИНАМИЧЕСКИЕ МАССИВЫ**

Динамический массив не может быть динамическим по нескольким измерениям, то есть выражение new T[n][m], где оба значения n и m определяются в процессе выполнения программы, не корректно. Но мы можем создать динамический массив, каждый элемент которого является встроенным массивом с размером, известным на стадии компиляции. Если M выражение, допустимое для определения размера массива, то следующая инструкция создает такой массив:

T(\*pa)[M] = **new** T[n][M];

Оператор new[] возвращает указатель на массив. Доступ к элементам такого массива будет осуществляться через выражение pa[i][j], в свою очередь pa[i] будет массив из M элементов типа T.

При использовании псевдонимов можно получить более лаконичный синтаксис.  
  
**using** I4 = **int**[4];

I4 \*pa = **new** I4[n];

Используя перегрузку оператора [] легко создать класс, который хранит данные в одномерном массиве, но при этом предоставляет интерфейс многомерного массива. Вот пример предельно упрощенного класса матрицы.

**template**<**typename** T>  
**class** MatrixView // 2D interface to a buffer  
{  
 T \* **const** m\_Buff;  
 **int** **const** m\_RowCount;  
 **int** **const** m\_ColCount;  
**public**:  
 MatrixView(T\* buff, **int** rowCount, **int** colCount)  
 : m\_Buff(buff)  
 , m\_RowCount(rowCount)  
 , m\_ColCount(colCount)  
 {}  
 T \***operator**[](**int** rowInd) **const** {  
 **return** m\_Buff + rowInd \* m\_ColCount;  
 }  
};  
  
**template**<**typename** T>  
**class** DynBuffer // buffer owner  
{  
 T\* **const** m\_Buff;  
**protected**:  
 T\* Buff() **const** { **return** m\_Buff; };  
 DynBuffer(**int** length) : m\_Buff(**new** T[length]{}) {}  
 ~DynBuffer() { **delete**[] m\_Buff; }  
 DynBuffer(**const** DynBuffer&) = **delete**;  
 DynBuffer& **operator**=(**const** DynBuffer&) = **delete**;  
};  
  
**template**<**typename** T>  
**class** MatrixSimple   
 : DynBuffer<T>, **public** MatrixView<T>  
{  
 **using** Buff = DynBuffer<T>;  
 **using** View = MatrixView<T>;  
**public**:  
 MatrixSimple(**int** rowCount, **int** colCount)  
 : Buff(rowCount \* colCount)  
 , View(Buff::Buff(), rowCount, colCount)  
 {}  
};

Вот пример использования:

MatrixSimple<**int**> mtx(3, 3);  
mtx[1][2] = 42; // первая строка, второй столбец

Более продвинутый класс матрицы может использовать специальный вложенный proxy-класс, представляющий строку, например, RowProxy, и индексатор будет возвращать экземпляр этого класса. Такой класс может, например, контролировать значение индекса, предоставлять функции-члены begin(), end(), etc. Аналогичное решение может быть и для столбцов.

# **ИСПОЛЬЗОВАНИЕ МАССИВОВ В ШАБЛОНАХ**

Тип массива можно использовать в качестве шаблонных аргументов и для специализации шаблонов классов.

Можно определить частичную специализацию шаблона класса для массивов, не задавая при этом размер массива, то есть для массивов «вообще». Для этого в качестве типа специализации надо использовать T[]. Конечно, можно определить частичную специализацию для массива с заданным размером. Вот пример.

// первичный шаблон  
**template** <**typename** T>  
**struct** U  
{  
 **const** **char\*** Tag() **const** { **return** "primary"; }  
};  
  
// частичная специализация для указателей  
**template** <**typename** T>  
**struct** U<T\*>  
{  
 **const** **char\*** Tag() **const** { **return** "pointer"; }  
};  
  
// частичная специализация для массивов  
**template** <**typename** T>  
**struct** U<T[]>  
{  
 **const** **char\*** Tag() **const** { **return** "array"; }  
};  
  
// частичная специализация для массивов с заданным размером  
**template** <**typename** T, **size\_t** N>  
structs U<T[N]>  
{  
 **const** **char\*** Tag() **const** { **return** "array[N]"; }  
};  
  
U<**int**> u1;  
U<**int\***> u2;  
U<**int**[]> u3;  
U<**int**[4]> u4;  
  
std::cout << u1.Tag() << ' ' << u2.Tag() << ' ' << u3.Tag() ' ' << u4.Tag();

Вывод:

primary pointer array array[N]

В стандартной библиотеке частичная специализация интеллектуального указателя std::unique\_ptr<> и std::shared\_ptr<> для массивов используется для управления жизненным циклом динамического массива.

Для программирования шаблонов, использующих массивы в качестве шаблонных аргументов, в стандартной библиотеке (заголовочный файл <type\_traits>) имеется несколько свойст типов: std::is\_array <>, std::extent<>,  std::rank<>,  std::remove\_extent<>. Вот примеры их использования (в примерах используется появившаяся в C++17 возможность использовать суффикс \_v вместо члена value):

std::cout <<  
 std::is\_array\_v<**int**[]> << ' ' <<  
 std::is\_array\_v<**int**[4]> << ' ' <<  
 std::is\_array\_v<**int**[4][8]> << ' ' <<  
 std::is\_array\_v<**int**> << '\n';  
std::cout <<   
 std::extent\_v<**int**[]> << ' ' <<  
 std::extent\_v<**int**[4]> << ' ' <<  
 std::extent\_v<**int**[4][8], 1> << ' ' <<  
 std::extent\_v<**int**> << '\n';  
std::cout <<  
 std::rank\_v<**int**[]> << ' ' <<  
 std::rank\_v<**int**[4]> << ' ' <<  
 std::rank\_v<**int**[4][8]> << ' ' <<  
 std::rank\_v<**int**> << '\n';

Вывод:

1 1 1 0

0 4 8 0

1 1 2 0

В качестве реального примера использования этих свойств типов приведем немного упрощенное определение перегруженного варианта шаблона функции std::make\_unique<> для массивов

**template** <**class** T,   
 **enable\_if\_t**<is\_array\_v<T> && extent\_v<T> == 0, **int**> s = 0>  
unique\_ptr<T> make\_unique(**size\_t** size) {  
 **using** **elem\_t** = **remove\_extent\_t**<T>; // тип элемента массива  
 **return** unique\_ptr<T>(**new** **elem\_t**[size]{});  
}

Шаблоны функций не поддерживают частичную специализацию, поэтому здесь используется техника, которая называется отключение шаблонов (template disabling). Этот шаблон будет отключен, то есть не будет конкретизироваться, для любых аргументов шаблона, тип которых отличается от T[]. Соответственно, перегруженный вариант std::make\_unique<> для аргументов шаблона остальных типов аналогичным способом будет отключен для T[].

# **СТАНДАРТНЫЕ АЛЬТЕРНАТИВЫ МАССИВАМ**

Стандартная библиотека предоставляет несколько классов (точнее шаблонов классов), которые рекомендуется использовать вместо массивов.

Вместо встроенных массивов рекомендуется использовать шаблон std::array<>. (Появился в C++11, см. [Josuttis].) Этот шаблон является объектной оберткой встроенного массива, он имеет два шаблонных параметра: тип элементов и размер. Размер должен быть известен на стадии компиляции, но в отличии от встроенного массива может быть нулевым. Вот пример:

std::array<**int**, 4> a{1, 2, 3, 4};

Этот шаблон поддерживает индексатор и традиционный интерфейс стандартного контейнера.

**for** (**int** i = 0; i < a.size(); ++i)  
{  
 std::cout << a[i] << ' ';  
}  
**for** (**auto** it = a.begin(); it != a.end(); ++it)  
{  
 std::cout << \*it << ' ';  
}  
**for** (**auto** t : a)  
{  
 std::cout << t << ' ';  
}

Вместо динамических массивов рекомендуется использовать std::vector<>. Этот шаблон хорошо известен программистам, подробно описан в литературе (стандартный контейнер №1), поэтому каких-то дополнительных подробностей можно не приводить.

Есть еще довольно специфический и не особо популярный шаблон std::valarray<>. Он позволяет эмулировать интерфейс многомерных массивов.

# **СПИСОК ЛИТЕРАТУРЫ**

Джосаттис, Николаи М. Стандартная библиотека C++: справочное руководство, 2-е изд.: Пер. с англ. — М.: ООО «И.Д. Вильямс», 2014.