

Массивы в C++



Основные термины

Для данного типа T , $T[\text{size}]$ есть тип «массив из size элементов типа T ». Элементы нумеруются (индексируются) от 0 до $\text{size}-1$.

Например:

```
double v[3];  
int a[32];
```

Количество элементов массива должно быть константным выражением.

Основные термины

Многомерные массивы описываются как массивы массивов.

Например:

```
int d[10][20];
```

d является массивом из 10 массивов по 20 целых.

Инициализаторы массивов

Начальные значения элементам массива можно присвоить, указав список значений.

Например:

```
int v1[]={ 1, 4, 6, 2};  
char v2[] = { 't', 'e', 'x', 't', 0 };
```

Не существует присваивания массиву, соответствующего описанному выше способу инициализации.

Доступ к элементам

Доступ к элементам массива осуществляется по имени массива и индексу элемента.

Например:

```
int v1[]={ 1, 4, 6, 2};  
    v1[0]=5;  
    cout<<v1[2]
```

Массивы как аргументы функций

Массив всегда неявно передаётся по ссылке. Это означает, что копии массива не создаётся и функция работает с альтернативным именем оригинального массива. Изменение элементов массива в функции приведёт к их изменению в оригинальном массиве.

При передаче массива в функцию размер массива (если он нужен в алгоритме) необходимо передавать явно. Функция не делает никаких предположений относительно размера массива. То есть она получает тип «массив элементов заданного типа», но размер остаётся неизвестен.

Примеры

Обнулить элементы массива

```
void fill(int arr[], int count)
{
    for (int i=0; i < count; ++i)
        arr[i]=0;
}
```


Последовательное заполнение

```
void fill_sequential
(int arr[], int count, int start_value)
{
    for (int i=0; i < count; ++i)
        arr[i]=start_value++;
}
```

Поиск заданного элемента

Постановка задачи: требуется определить, присутствует ли заданный элемент (ключ) в массиве. Если элемент присутствует – вернуть его индекс.

Поиск заданного элемента

//Поиск ключа в массиве

//Функция возвращает индекс искомого элемента

//В случае отсутствия искомого элемента

//возвращает -1.

```
int find_1(int arr[], int count, int key ) {  
    for (int i=0; i < count; ++i)  
        if ( arr[i] == key)  
            return i;  
  
    return -1;  
}
```

Поиск заданного элемента. Двоичный поиск

Постановка задачи сохраняется - требуется определить, присутствует ли заданный элемент (ключ) в массиве. Если элемент присутствует – вернуть его индекс.

Однако, есть возможность значительно ускорить поиск, если ввести дополнительное предусловие к исходному массиву: данные в нём должны быть упорядочены.

Будем полагать, что в исходном массиве данные упорядочены по возрастанию.

Поиск заданного элемента. Двоичный поиск

```
int bin_search(int arr[],int count,int key)
{
    int l=0;                // нижняя граница
    int u=count-1;          // верхняя граница
    while ( l <= u ) {
        int m = (l+u)/2;
        if (arr[m] == key) return m;
        if (arr[m] < key) l = m + 1;
        if (arr[m] > key) u = m -1;
    }
    return -1;
}
```

Указатели в языке C++



Определение указателя

```
int value;
```

```
int value;
```

```
int *p_value = &value;
```

Определение указателя

```
int value;
```

```
int value;
```

```
int *p_value = &value;
```

```
*p_value = 1;
```

```
cout << "Value=" << *p_value;
```


Указатели на массивы

```
int v[] = { 1,2,3,4 };
```

```
int *p1 = v;      // Указатель на первый элемент
```

```
int *p2 = &v[0]   // Указатель на первый элемент
```

```
int *p3 = &v[4]   // Указатель на элемент, следующий за последним
```

Обнуление элементов массива

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

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

Указатели и константы

```
const int *p1; // указатель на константу типа int
```

```
int const *p2; // указатель на константу типа int
```

```
int *const p3; // константный указатель на объект типа int
```

```
const int *const p4; // константный указатель на константу  
                      // типа int
```

```
int const *const p4; // константный указатель на константу  
                      // типа int
```

Указатели на строки

```
void f() {  
    char *p="text";  
    p[3] = 'a';    // ОШИБКА  
}
```

Указатели на строки

```
void f() {  
    char *p="text";  
    p[3] = 'a';    // ОШИБКА  
}
```

```
const char *access(int i) {  
    ...  
    return "access denied";  
}
```

Указатели на строки

```
void f() {  
    char *p="text";  
    p[3] = 'a';    // ОШИБКА  
}
```

```
const char *access(int i) {  
    ...  
    return "access denied";  
}
```

```
const char src[] = "Строка, которую надо скопировать";  
char dst[sizeof src];  
const char *p_src = src;  
char *p_dst = dst;  
while (*p_dst++ = *p_src++)  
    ;
```

Массивы как параметры функции

```
void foo(int arr[]) {  
    int size = sizeof(arr); // Размер в байтах указателя на int  
    ...  
}  
  
int main() {  
    int arr[10];  
    foo(arr);  
}
```

Указатели на функции

```
void error(int i);  
void (*p)(int);  
p = &error;  
(*p)(1);
```


Указатели на функции

```
void error(int i);  
void (*p)(int);  
p = &error;  
(*p)(1);
```

```
void error(int i);  
void (*p)(int);  
p = error;  
p(1);
```

Указатели и структуры

```
struct vec2 { double x, y; };  
vec2 dir;  
vec2 *p_dir;  
(*p_dir).x = 0;  
p_dir->x = 0;
```

Динамическое распределение памяти

Динамическое распределение памяти

Время жизни именованного объекта определяется его областью видимости. С другой стороны часто возникает необходимость в создании объектов, которые существуют вне зависимости от области видимости, в которой они созданы.

Динамическое распределение памяти

Одним из решений в данном случае может являться использование глобальных объектов, время жизни которых равно времени жизни программы. Однако глобальные объекты не решают других задач:

1. Возможности создавать объект только тогда, когда он необходим (например, при возвращении объекта как результата работы функции)
2. Возможности создавать массив объектов заранее неизвестного размера, т.е. в том случае, когда размер массива становится известен в ходе выполнения программы.

Динамическое распределение памяти

Для решения таких задач лучше всего подходит механизм динамического создания объектов из заранее выделенной области свободной памяти. Такая память называется «кучей» (heap) и выделяется операционной системой и системными библиотеками при запуске программы. Это неименованная область памяти, которая при запуске принадлежит операционной системе (или программе) и не принадлежит никакому конкретному объекту в программе.

Динамическое распределение памяти

Для работы с динамической памятью («кучей») используются операторы **new** и **delete**.

Оператор **new** позволяет динамически выделить часть памяти из «кучи» и получить указатель на выделенный участок памяти.

У оператора **new** есть две формы: **new** и **new[]**.

Динамическое распределение памяти

Первая форма позволяет распределить память для одного объекта заданного типа и имеет следующий синтаксис:

T^* new T

Пример:

```
Circle *p;  
p = new Circle;
```


Динамическое распределение памяти

Вторая форма оператора new позволяет выделить память для распределения массива объектов:

`T* new T[size]`

Оператор `new[]` выделяет непрерывный участок памяти для `size` объектов типа `T` и возвращает указатель на выделенный участок памяти.

Пример:

```
Circle *array;  
array = new Circle[nCircles];
```

Динамическое распределение памяти

Обе формы оператора `new` обеспечивают вызов конструкторов создаваемых объектов пользовательских типов.

Для объектов, созданных в динамической памяти неявной инициализации не производится, то есть в случае, если явно не указано начальное значение для создаваемого объекта, то начальное значение будет не определено («мусор»).

Динамическое распределение памяти

Для освобождения ранее выделенной оператором `new` памяти используется оператор `delete`. Он возвращает ранее выделенную память обратно в «кучу» с тем, чтобы её можно было использовать для создания других объектов.

У оператора `delete` также существует две формы: `delete` и `delete[]`.

Динамическое распределение памяти

Первая форма используется для освобождения памяти, распределённой оператором new:

Пример:

```
Circle *p;  
p = new Circle;  
...  
delete p;
```

Динамическое распределение памяти

Вторая форма используется для освобождения памяти, распределённой оператором `new[]` :

Пример:

```
Circle *array;  
array = new Circle[nCircles];  
...  
delete[] p;
```

Динамическое распределение памяти

Обе формы оператора `delete` обеспечивают вызов деструкторов создаваемых объектов пользовательских типов.

Память, выделенная оператором `new` должна освобождаться только оператором `delete`, а память, выделенная оператором `new[]` должна освобождаться только оператором `delete[]`. Нарушение этого правила является ошибкой и ведёт к непредсказуемому поведению программы.

Динамическое распределение памяти

В случае, если в операторе new запросить массив нулевого размера (с нулевым числом элементов), то оператор new вернёт уникальный адрес памяти, но использовать его для записи или чтения будет нельзя, поскольку реально зарезервировано ноль байт.

Динамическое распределение памяти

Повторное применение оператора delete к одному и тому же указателю является ошибкой и приведёт к непредсказуемому поведению программы.

Динамическое распределение памяти

Применение оператора `delete` к нулевому указателю (указателю, значением которого является ноль), не приводит к выполнению каких-либо действий. То есть это «пустая» операция, которая не приведёт к ошибке. (Указанное правило верно для встроенных типов и пользовательских типов, у которых не перегружены операторы `new` и `delete`).

Динамическое распределение памяти

В старых программах на С для выделения памяти из кучи используются функции `malloc` и `free`. Их не следует использовать в программах на С++, поскольку они не обеспечивают безопасности в отношении типов, не вызывают деструкторов и имеют ряд других недостатков.

Смешивать управление памяти по `new/delete` и `malloc/free` недопустимо. То есть, память, выделенную по одному из этих механизмов нельзя освобождать другим. Это является ошибкой и ведёт к непредсказуемому поведению программы.

`nullptr`

Предпочитайте `nullptr` значениям `0` и `NULL`

`std::unique_ptr`

Используйте `std::unique_ptr` для управления ресурсами путем исключительного владения.

Интеллектуальные указатели `std::unique_ptr` воплощают в себе семантику исключительного владения. Ненулевой `std::unique_ptr` всегда владеет тем, на что указывает.

Перемещение `std::unique_ptr` передает владение от исходного указателя целевому. (Исходный указатель при этом становится нулевым.) Копирование `std::unique_ptr` не разрешается, так как если вы можете копировать `std::unique_ptr`, то у вас будут два `std::unique_ptr`, указывающих на один и тот же ресурс, и каждый из них будет считать, что именно он владеет этим ресурсом (а значит, должен его уничтожить). Таким образом, `std::unique_ptr` является только перемещаемым типом.

`std::unique_ptr`

При деструкции ненулевой `std::unique_ptr` освобождает ресурс, которым владеет. По умолчанию освобождение ресурса выполняется с помощью оператора `delete`, примененного ко встроенному указателю в `std::unique_ptr`.

- `std::unique_ptr` представляет собой маленький, быстрый, предназначенный только для перемещения интеллектуальный указатель для управления ресурсами с семантикой исключительного владения.
- По умолчанию освобождение ресурсов выполняется с помощью оператора `delete`, но могут применяться и пользовательские деструкторы. Деструкторы без состояний и указатели на функции в качестве деструкторов увеличивают размеры объектов `std::unique_ptr`.
- Интеллектуальные указатели `std::unique_ptr` легко преобразуются в интеллектуальные указатели `std::shared_ptr`.

`std::shared_ptr`

Используйте `std::shared_ptr` для управления ресурсами путем совместного владения.

Объект, доступ к которому осуществляется через указатели `std::shared_ptr`, имеет время жизни, управление которым осуществляется этими указателями посредством совместного владения. Никакой конкретный указатель `std::shared_ptr` не владеет данным объектом. Вместо этого все указатели `std::shared_ptr`, указывающие на него, сотрудничают для обеспечения гарантии, что его уничтожение произойдет в точке, где он станет более ненужным.

Указатель `std::shared_ptr` может сообщить, является ли он последним указателем, указывающим на ресурс, с помощью счетчика ссылок.

`std::shared_ptr`

Размер `std::shared_ptr` в два раза больше размера обычного указателя, поскольку данный интеллектуальный указатель содержит обычный указатель на ресурс и другой обычный указатель на счетчик ссылок.

Инкремент и декремент счетчика ссылок должны быть атомарными, поскольку могут присутствовать одновременное чтение и запись в разных потоках.

Для создания указателя используйте функцию `std::make_shared` вместо `new`.