Указатели. Динамическая память

Максим Бакиров
C++ - Разработчик в Яндекс

Максим Бакиров

О спикере:

- В C++ разработке с 2017 года
- С 2019 года работает в команде разработки Яндекс Браузера

Вспоминаем прошлое занятие

Вопрос: Как узнать тип значения?

Вспоминаем прошлое занятие

Вопрос: Как узнать тип значения?

Ответ: С помощью оператора **typeid**

Вспоминаем прошлое занятие

Вопрос: Как правильно вызвать **typeid**?

Вспоминаем прошлое занятие

Вопрос: Как правильно вызвать **typeid**?

Ответ: typeid(<значение>).name()

Вспоминаем прошлое занятие

Вопрос: Как объявить указатель на тип **float**?

Вспоминаем прошлое занятие

Вопрос: Как объявить указатель на тип **float**?

Ответ: float *

Вспоминаем прошлое занятие

Вопрос: Как разыменовать указатель?

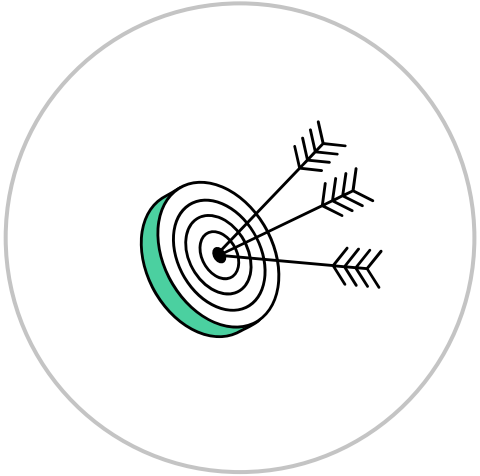
Вспоминаем прошлое занятие

Вопрос: Как разыменовать указатель?

Ответ: С помощью оператора разыменования * (звёздочка)

Цели занятия

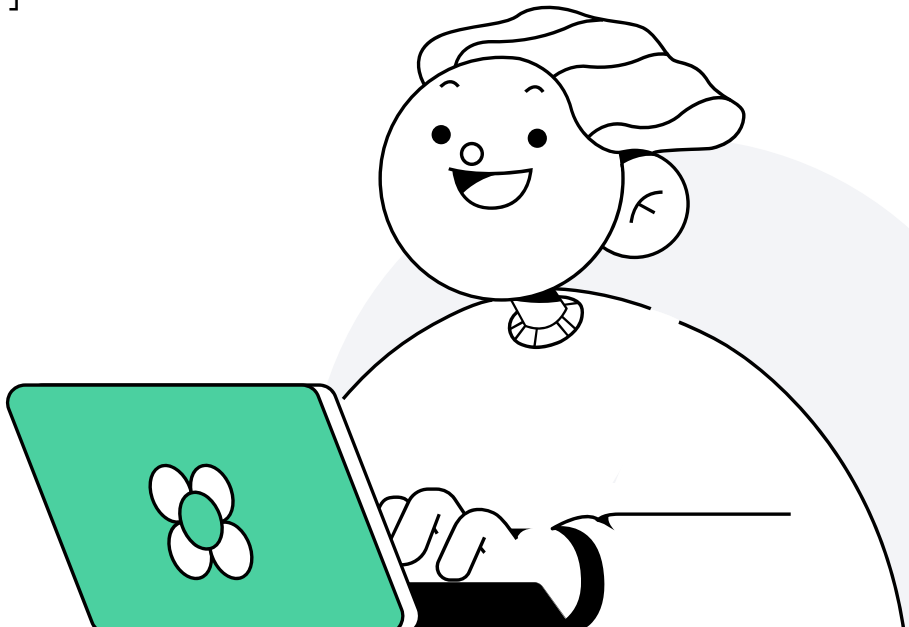
- Разберёмся с тем, как выделять и очищать динамическую память
- Познакомимся с одномерными динамическими массивами
- Узнаем, каковы особенности динамической памяти по сравнению с автоматической
- Выясним, как работать с двумерными динамическими массивами



План занятия

1. Динамическая память: malloc, calloc и free
2. Динамическая память: new и delete
3. Динамическая память: new[ ] и delete[ ]
4. Динамическая память в функциях
5. Двумерные динамические массивы
6. Итоги
7. Домашнее задание

*Нажми на нужный раздел для перехода



Динамическая память. Функции malloc, calloc и free

Динамическая память

Динамическая память — это память, которой в рамках программы управляет программист. Он должен **запросить** эту память у операционной системы, затем **использовать** её и в конце обязан **очистить** эту память

Функция malloc

Функция malloc объявлена в библиотеке **cstdlib**. Для использования функции необходимо подключить эту библиотеку. Подключение осуществляется с помощью препроцессорной директивы **include** так же, как мы подключаем библиотеку **iostream** для использования вывода на консоль (std::cout)

```
#include <cstdlib>
int main(int argc, char** argv)
{
    // мы можем использовать функцию malloc
}
```

Функция malloc

Если подключить библиотеку iostream, то отдельно подключать cstdlib не надо — она уже включена в iostream

```
#include <iostream>
int main(int argc, char** argv)
{
    // тоже можем использовать функцию malloc
}
```

Что делает malloc

Функция malloc **запрашивает** у операционной системы определённое **количество** байт. Это количество вы передаёте как аргумент при вызове функции malloc. Без аргумента вызвать её не получится

```
#include <cstdlib>
int main(int argc, char** argv)
{
   malloc(20); // запросили у операционной системы 20 байт памяти
}
```

Что делает malloc

После вызова функция malloc возвращает нам указатель на область памяти, которую выделила нам операционная система.

Здесь важно, что возвращаемым типом функции malloc является **void*** — это специальный тип указателя, который показывает, что он не знает, на значения какого типа он указывает

```
#include <cstdlib>
int main(int argc, char** argv)
{
    void * ptr = malloc(20); // сохранили указатель на 20 байт выделенной памяти в переменную ptr
}
```

Что делает malloc

Чтобы иметь возможность пользоваться выделенной памятью, например, записать в неё несколько целых чисел, нам необходимо привести полученный нами указатель к указателю требуемого типа. В случае с целыми числами это будет int *

```
#include <cstdlib>
int main(int argc, char** argv)
{
    int * int_ptr = static_cast<int *>(malloc(20)); // int_ptr позволит нам работать с целыми числами
}
```

Зачем используют malloc

В подавляющем большинстве случаев malloc служит для выделения памяти, которая будет использована как **динамический массив**. Он отличается от локального тем, что его размер определяется на **этапе выполнения** программы, а не на этапе её компиляции.

Но в аргументах функции malloc мы указываем не количество **элементов** массива, а количество **байт**. Поэтому в 20 байт выделенной памяти поместится всего 5 целых чисел — ведь каждое целое число занимает 4 байт

Динамический массив

Особенность динамического массива: вы можете создавать массив, размер которого вам **заранее неизвестен**. Например, вы можете получать размер массива от пользователя.

Кажется, что сложно каждый раз считать, сколько нам байт надо под, например, 29 элементов массива. Но есть способ не считать это самим, а заставить программу посчитать за нас

Как вы думаете, как это сделать? Какой нужен оператор?
Напишите в чат

Динамический массив

Нужно использовать оператор **sizeof**.

Вместо того чтобы самим умножать 29 на 4, вы напишете: 29 * **sizeof(int)**.
Таким образом легко выделять память под элементы разных типов, достаточно изменить тип в операторе **sizeof**

```
#include <cstdlib>
int main(int argc, char** argv)
{
    int * int_ptr = static_cast<int *>(malloc(29 * sizeof(int))); // в массив
int_ptr поместится 29 целых чисел
}
```

Потренируемся

```
#include <cstdlib>
int main(int argc, char** argv)
{
    //что написать, чтобы создать динамический массив на 13 элементов double?
}
```

Потренируемся

```
#include <cstdlib>
int main(int argc, char** argv)
{
    double * dbl_ptr = static_cast<double *>(malloc(13 * sizeof(double)));
}
```

Очищение

Динамически выделенную память после использования надо **очистить**. Для этого используется функция **free**. Эта функция принимает в качестве аргумента указатель на динамически выделенную память.

Самая большая сложность в использовании функции free заключается в том, чтобы не забыть её вызвать. Если вы этого не сделаете, то произойдёт **утечка памяти**

```
#include <cstdlib>
int main(int argc, char** argv)
{
    int * int_ptr = static_cast<int *>(malloc(20 * sizeof(int))); // создали
динамический массив на 20 целых чисел
    int_ptr[0] = 65; // использовали этот массив
    free(int_ptr); // освобождаем память
}
```

Функция calloc

Одна из особенностей функции malloc заключается в том, что она никак не изменяет значения в той памяти, указатель на которую она вам возвращает. То есть она **не инициализирует** её. Это значит, что там потенциально лежит «мусор» — случайные значения.

Если вам нужно, чтобы все значения в памяти, которая для вас выделяется, были **инициализированы нулями**, то вам следует воспользоваться функцией **calloc**

Функция calloc

Функция **calloc** тоже выделяет память, как и функция malloc, однако все значения гарантированно будут равны нулю.

Ещё одно отличие функции calloc от malloc заключается в том, что она принимает на вход два аргумента: **количество** элементов и **размер** одного элемента.
То, что в malloc умножается друг на друга, в calloc передаётся в виде аргументов по отдельности

```
#include <cstdlib>
int main(int argc, char** argv)
{
    int * int_ptr = static_cast<int *>(calloc(20, sizeof(int))); // в массив
int_ptr поместится 20 целых чисел
}
```

Проверка на внимательность

В остальном функция calloc работает так же, как и функция malloc.

Вопрос: мы хотим создать и использовать динамический массив на 15 элементов типа float и инициализировать его нулями, а последнему элементу присвоить значение 43.8. Что здесь не так?

```
#include <cstdlib>
int main(int argc, char** argv)
{
    int size = 15;
    float * flt_ptr = static_cast<int *>(calloc(size * sizeof(float)));
    flt_ptr[size] = 43.8;
    return 0;
}
```

Проверка на внимательность

В остальном функция calloc работает так же, как и функция malloc.

Вопрос: мы хотим создать и использовать динамический массив на 15 элементов типа float и инициализировать его нулями, а последнему элементу присвоить значение 43.8. Что здесь не так?

```
#include <cstdlib>
int main(int argc, char** argv)
{
    int size = 15;
    float * flt_ptr = static_cast<int *>(calloc(size * sizeof(float)));
     // приведение указателя не к тому типу у функции calloc 2 аргумента
    flt_ptr[size] = 43.8;   // выход за границы массива
    return 0;               // забыли очистить память
}
```

Правильный вариант

В остальном функция calloc работает так же, как и функция malloc.

Вопрос: мы хотим создать и использовать динамический массив на 15 элементов типа float и инициализировать его нулями, а последнему элементу присвоить значение 43.8. Что здесь не так?

```
#include <cstdlib>
int main(int argc, char** argv)
{
    int size = 15;
    float * flt_ptr = static_cast<float *>(calloc(size, sizeof(float)));
    flt_ptr[size - 1] = 43.8;
    free(flt_ptr);
    return 0;
}
```

Заключение

Функции malloc, calloc и free являются наиболее старыми инструментами для работы с динамической памятью. Однако они не слишком удобны в использовании, поэтому мы познакомимся с более современным вариантом

Перерыв

Динамическая память. Операторы new и delete

2

Оператор new

Для создания динамических переменных, содержащих одно значение, можно использовать оператор **new**. Он возвращает указатель на выделенный участок динамической памяти.

Синтаксис его выглядит следующим образом: **new** **<тип>**

Создадим целочисленное значение, которое будет находиться в динамической памяти:

```
int main(int argc, char** argv)
{
    int* var = new int;// выделяем динамическую память для размещения там одного целого числа
    *var = 50;          // пользуемся указателем на динамическую память
}
```

Оператор delete

Созданные таким образом переменные, так как они находятся в динамической памяти, необходимо самостоятельно очищать. Для этого используется оператор delete. Пользоваться им очень просто: когда память стала нам больше не нужна, необходимо вызвать оператор delete и после него написать указатель на динамическую память

```
int main(int argc, char** argv)
{
    int* var = new int;         // выделяем динамическую память для размещения там одного целого числа
    *var = 50;                  // пользуемся указателем на динамическую память
    delete var;                 // очищаем динамическую память
}
```

Динамическая память. Операторы new[] и delete[]

Оператор new[]

Для создания динамических массивов существует гораздо более удобный инструмент — оператор **new** для массивов.

Синтаксис его выглядит следующим образом: new <тип>[<количество>]

Выделим массив на 20 элементов типа **int**

```
int main(int argc, char** argv)
{
    int size = 20;
    int* int_ptr = new int[size];
}
```

Оператор new[ ]. Список инициализации

Использовать такой оператор гораздо проще, чем malloc или calloc.
Более того, он предоставляет возможность инициализировать массив с помощью **списка инициализации** — прямо как локальный. Однако для этого нужно знать длину массива на этапе компиляции, потому что список **инициализации** нельзя составить динамически

```
int main(int argc, char** argv)
{
    int* int_ptr = new int[5] {1, 2, 3, 4, 5}; // инициализируем массив значениями
}
```

Оператор new[ ]. Инициализация нулями

Если нужно инициализировать массив нулями, как в функции calloc, то для этого нужно просто поставить пустые скобки после квадратных

```
int main(int argc, char** argv)
{
    int* int_ptr = new int[5](); // инициализируем массив нулями
}
```

Оператор delete[ ]

Динамически выделенные массивы всё ещё нужно очищать. Для этого существует **оператор delete** для массивов. Синтаксис его очень простой:

delete[] <указатель на массив>

```
int main(int argc, char** argv)
{
    int* int_ptr = new int[5];   // выделяем динамический массив на 5 элементов
    delete[] int_ptr;            // очищаем память
}
```

Динамическая память в функциях

4

Динамическая память в функциях

Важно понимать, что локальные (автоматические) переменные очищаются, когда заканчивается блок кода, в рамках которого они были объявлены.

Это не касается динамической памяти, которая будет ассоциирована с программой до тех пор, пока её не очистят

Динамическая память в функциях

Посмотрите на этот блок кода. Здесь в рамках функции create_arr создаётся массив arr, который затем возвращается из функции. Ведь массив — это указатель. Скажите, arr — это какой массив?

```
int* create_arr()
{
    int arr[10];
    return arr;
}
int main(int argc, char** argv)
{
    int* external_arr = create_arr();
}
```

Динамическая память в функциях

arr — это локальный (автоматический) массив. Это значит, что память, выделенная под массив arr в рамках функции create_arr, будет помечена как свободная при выходе из функции create_arr, и дальнейшее использование указателя на эту память вне функции create_arr будет иметь непредсказуемые последствия

```
int* create_arr()
{
    int arr[10];
    return arr;
}
int main(int argc, char** argv)
{
    int* external_arr = create_arr();
}
```

Динамическая память в функциях

Чтобы создать массив в функции и отдать его вызывающему коду, чтобы он смог им пользоваться, нужно использовать динамические массивы. Такой массив можно будет использовать в функции main. Скажите, что не так с этим кодом?

```
int* create_arr()
{
    return new int[10];
}
int main(int argc, char** argv)
{
    int* external_arr = create_arr();
    return 0;
}
```

Динамическая память в функциях

Динамическую память мы должны чистить сами. Здесь не хватает вызова оператора `delete[]`

```
int* create_arr()
{
    return new int[10];
}
int main(int argc, char** argv)
{
    int* external_arr = create_arr();
    delete[] external_arr;
    return 0;
}
```

Напишем программу, которая создаёт массив указанного пользователем размера и заполняет его целыми числами по порядку

Готовый пример кода

Двумерные динамические массивы

5

Создание двумерного динамического массива

Двумерные динамические массивы отличаются от двумерных автоматических массивов. Чтобы создать двумерный динамический массив, необходимо:

1. Создать одномерный динамический массив с указателями нужного типа (например, int *)
2. Для каждого указателя создать одномерный динамический массив нужного типа
3. Положить указатели на массивы, созданные в п. 2 в массив, созданный в п. 1

Создание двумерного динамического массива

Посмотрим на примере — создадим двумерный динамический массив, содержащий 3 строки и 4 столбца

```
int main(int argc, char** argv)
{
    int rows = 3, cols = 4;
    int** arr = new int*[rows]; // создаём массив указателей (int*). Сам массив будет иметь тип int**
    for (int i = 0; i < rows; i ++)
    {
        arr[i] = new int[cols]; // для каждой ячейки массива arr создаём массив целых чисел и кладём указатель на вновь созданный массив в эту ячейку
    }
    arr[1][2] = 3;              // используем двумерный массив
    return 0;
}
```

Очищение двумерного динамического массива

Чтобы очистить двумерный динамический массив, необходимо:

1. В цикле пройти по массиву указателей и очистить каждый указатель отдельно. Потому что каждый указатель указывает на свой кусок динамической памяти
2. После этого очистить указатель на массив указателей

Очищение двумерного динамического массива

```
int main(int argc, char** argv)
{
    int rows = 3, cols = 4;
    int** arr = new int*[rows]; // создаём массив указателей (int*). Сам массив
будет иметь тип int**
    for (int i = 0; i < rows; i++)
    {
        arr[i] = new int[cols]; // для каждой ячейки массива arr создаём массив
целых чисел и кладём указатель на вновь созданный массив в эту ячейку
    }
    arr[1][2] = 3;              // используем двумерный массив
    for (int i = 0; i < rows; i++)
    {
        delete[] arr[i];        // очищаем каждый подмассив отдельно
    }
    delete[] arr;               // очищаем верхнеуровневый массив указателей
    return 0;
}
```

Итоги

Итоги занятия

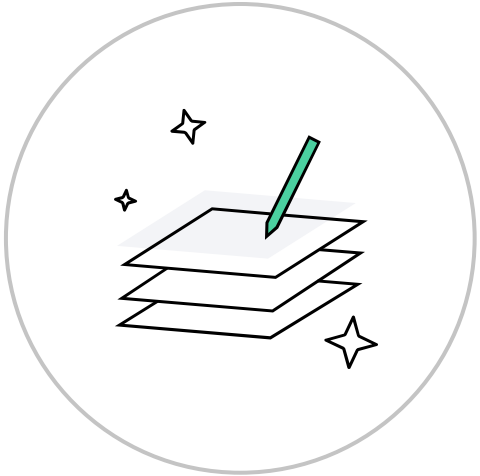
Сегодня мы:

1. Разобрались с тем, как выделять и очищать динамическую память
2. Познакомились с одномерными динамическими массивами
3. Узнали, каковы особенности динамической памяти по сравнению с автоматической
4. Выяснили, как работать с двумерными динамическими массивами

Домашнее задание

Давайте посмотрим ваше домашнее задание

1. Вопросы по домашней работе задавайте в чате группы
2. Задачи можно сдавать по частям
3. Зачёт по домашней работе ставят после того, как приняты все задачи



Дополнительные материалы

- Динамическое выделение памяти
- Двумерные динамические массивы

Задавайте вопросы и пишите отзыв о лекции

Максим Бакиров
C++ - Разработчик в Яндекс